目錄

Part I: 目录与翻译注记(Table of contents and Notes)

Part II: 文档内容(Documentation contents)

Spring MVC 4.2.4.RELEASE 中文文档

build passing issues 17 open issues 9 closed



本项目翻译的是Spring MVC官方4.2.4.RELEASE版本的文档，包含原文档第21章Spring MVC部分的全部内容。译文尽力于准确传达原意，其次兼顾译文的流畅自然。至于风格和质感，则仍在努力。希望它能为读者带来查阅、学习的价值，自己时不时翻之，仍有收获。

目前多数章节的翻译已完成，剩余部分章节仍在进行。文档仍在维护状态，主要还有译文细化、术语定义、翻译规范、内容、主页修缮、自动化部署等工作可做，issues这里有一些有意思的idea。翻译过程中遇到值得探讨的翻译问题、取舍及最终解决方案，读者可见翻译注记。

本翻译初始只是自我学习需要，逐渐完善后才有坚持完成的执念。陈丹青在《木心谈木心》的后记中，讲到他犹豫于出版木心先生这本私房话的心境。为本译文做推广、宣传伊始，我也开始面对我的读者，读之，感觉真挚感动。不敢自比木心，我在我的风中等消息。

——2016年6月28 交房租日，8月28日 完成自动化部署后补稿

中文文档地址

- 主站：mvc.linesh.tw（速度和稳定性更好，样式也与原文档一致）
- 国外Gitbook原站

原文地址

Spring MVC 4.2.4.RELEASE Documentation

其他相关翻译项目

为了对目前Spring MVC部分文档翻译现状有个大致的了解，可以“Spring MVC 中文 文档 翻译”作为关键词，浏览其在google和baidu上前6页的搜索结果。其中以下项目值得留意，前两个均或完全或部分地翻译了Spring MVC部的内容，可供参考；后面三个项目未涉及Spring MVC部分的翻译。

项目	作者	项目 Github	描述
Spring框架参考文档	一个团队	Github	该项目规模较大、参与人数较多。翻译内容是Spring 4.1.3.RELEASE版本全部文档，其中MVC部分的文档也翻译了一大半。其项目主页保留了与原生Spring文档较一致的样式，很不错
Spring Framework 2.5翻译计划	满江红机构	-	感谢dsliu在gitbook上给我提供此版译本链接。译本是整个MVC 2.5.2版的全部文档，其中MVC的部分同样齐全
Spring Framework 4.x参考文档	waylau	Github	翻译了Spring文档的简介、新特性和容器IOC部分
Spring Framework 4.x中文翻译	sunrh	Github	翻译了Spring文档的简介、新特性和容器IOC部分
Spring 中文文档3.1	wizardforcel	-	主页已标记废弃的项目。楼主BIO是专注单身二十年，言语间竟有一种大学宿友~~不是说我的宿友~~的即视感

友情链接

这个译本我在国内的多个站点均发表过一篇相同的推广文章，如OSC/CSDN/Iteye/博客园/掘金/v2ex/segmentfault/Githuber等。除了交付的译文本身外，还聊~~瞎扯~~了一些其他的东西。同时，关于这个翻译文档的创始、管理及自动化部署等方面，我也已将其总结成为文章。此二篇文章是对这个项目的完整记录，均发布在我的博客上，有兴趣的读者可以前往阅读。后来我又做了一些主页样式上的迁移、自动化了一些构建前文档预处理的工作，还有一些代码的重构。这部分未做记录，但代码和部署构架方面我十分满意，其精华在 `package.json` 和构建脚本中。

- Spring MVC官方文档翻译稿发布
- 我是如何进行Spring MVC文档翻译项目的环境搭建、项目管理及自动化构建工作的

联系方式

阅读过程中的任何想法、建议、吐槽、强迫症~~不给译者狂点100个赞就浑身不舒服~~、觉得赞、觉得不赞，无论关于翻译、技术、样式等，对我来说很有意义~~啊我这文风竟有一种安妮宝贝般的性冷淡感~~！你可以通过以下的方式联系作者我：

- 来Github点赞 ~~被消费一个~~ Star 780
- 在Gitbook讨论里 给我留言
- 给这个项目提 issue
- 给这个项目提 pull request
- 邮箱：linesh.simpcity@gmail.com

License

MIT License

贡献者 Contributor

感谢那些让这个项目变得更好的人们。

吕立青 Sun SongWang 易枭寒 xcatliu HeartUnchange

目录

外延丰富/有业务含义的术语翻译：直译、保留原文、词汇表

一些术语难翻的点可能有以下各个方面，比如：

- 具有宽泛的外延/内涵/比喻义/象征义/指代义（比如request/response，有时指一个HTTP层面的请求，有时指一个应用层级的请求，有时又特指一个请求对象，有时又具体到特定类型 `HttpServletRequest` 请求对象中的内容或请求头等，又比如mapping，可以动词映射，也可以是名词什么什么映射，有时它还可以省略映射的目标或源对象)
- 具有相对自治、完整的业务含义（比如scope、action、bean、flashmap、multipart、session/conversation等），而该业务领域在中文技术世界又尚无对应词汇

目前大致的处理方式是：

- 对于其外延义尚可翻（只是中文语意承载量仍然不足）的单词，采用 直译+词汇表 的翻译。词汇表即是鼠标划过停留时，会显示额外的提示信息，在提示里更详细地解释这个词在英文中的完整含义）
- 对于业务含义较完备，且深刻体现在代码或命名中实在没法翻的单词，采用 保留原文+词汇表 的方式翻译

未决翻译

- locales：本地化？地区？
- multipart：多部分（这是中文）？多路？
- flashmap：我彻底地慌了。不知道怎么翻好，可能需要查一下TCP/IP这一块的名词，看看有没有什么专业字词能够体现、概括的

TODOLIST

- 翻译怎么样算好？具体到技术翻译这个上下文，全部照翻为好？谁都能意识到要适当变通，可变通的度是多少？如果要求的是字字都有对应译，还要符合中文风格，那此种翻译岂非只在寻求一个可能存在的“解”而失去任何可能的发挥？如果要求的是信达的同时可以体现风格，强调内在和谐胜于字句适配，那么风格的适当与否评价标准又在哪里？译者和作者的关系是怎样的？

21.1 Spring Web MVC框架简介

Spring的模型-视图-控制器（MVC）框架是围绕一个 `DispatcherServlet` 来设计的，这个Servlet会把请求分发给各个处理器，并支持可配置的处理器映射、视图渲染、本地化、时区与主题渲染等，甚至还能支持文件上传。处理器是你的应用中注解了 `@Controller` 和 `@RequestMapping` 的类和方法，Spring为处理器方法提供了极其多样灵活的配置。Spring 3.0以后提供了 `@Controller` 注解机制、 `@PathVariable` 注解以及一些其他的特性，你可以使用它们来进行RESTful web站点和应用的开发。

> “对扩展开放”是Spring Web MVC框架一个重要的设计原则，而对于Spring的整个完整框架来说，其设计原则则是“对扩展开放，对修改闭合”。
>
> Spring Web MVC核心类库中的一些方法被定义为 `final` 方法。作为开发人员，你不能覆写这些方法以定制其行为。当然，不是说绝对不行，但请记住这条原则，绝大多数情况下不是好的实践。
>
> 关于该原则的详细解释，你可以参考Seth Ladd等人所著的“深入解析Spring Web MVC与Web Flow”一书。相关信息在第117页，“设计初探（A Look At Design）”一节。或者，你可以参考：
>
> - Bob Martin所写的“开闭原则（The Open-Closed Principle）”（PDF）
>
> 你无法增强Spring MVC中的 `final` 方法，比如 `AbstractController.setSynchronizeOnSession()` 方法等。请参考10.6.1 理解AOP代理一节，其中解释了AOP代理的相关知识，论述了为什么你不能对 `final` 方法进行增强。

在Spring Web MVC中，你可以使用任何对象来作为命令对象或表单返回对象，而无须实现一个框架相关的接口或基类。Spring的数据绑定非常灵活：比如，它会把数据类型不匹配当成可由应用自行处理的运行时验证错误，而非系统错误。你可能会为了避免非法的类型转换在表单对象中使用字符串来存储数据，但无类型的字符串无法描述业务数据的真正含义，并且你还需要把它们转换成对应的业务对象类型。有了Spring的验证机制，意味着你再也不需这么做了，并且直接将业务对象绑定到表单对象上通常是更好的选择。

Spring的视图解析也是设计得异常灵活。控制器一般负责准备一个 `Map` 模型、填充数据、返回一个合适的视图名等，同时它也可以直接将数据写到响应流中。视图名的解析高度灵活，支持多种配置，包括通过文件扩展名、 `Accept` 内容头、bean、配置文件等的配置，甚至你还可以自己实现一个视图解析器 `ViewResolver` 。模型（MVC中的M，model）其实是一个 `Map` 类型的接口，彻底地把数据从视图技术中抽象分离了出来。你可以与基于模板的渲染技术直接整合，如JSP、Velocity和Freemarker等，或者你还可以直接生成XML、JSON、Atom以及其他多种类型的内容。 `Map` 模型会简单地被转换成合适的格式，比如JSP的请求属性（attribute），一个Velocity模板的模型等。

21.1.1 Spring Web MVC的新特性

> Spring Web Flow
>
> Spring Web Flow (SWF) 意在成为web应用中的页面流(page flow)管理中最好的解决方案。
>
> SWF在Servlet环境和Portlet环境下集成了现有的框架，如Spring MVC和JSF等。如果你的业务流程有一个贯穿始终的模型，而非单纯分立的请求，那么SWF可能是适合你的解决方案。
>
> SWF允许你将逻辑上的页面流抽取成独立可复用的模块，这对于构建一个web应用的多个模块是有益的。that guide the user through controlled navigations that drive business processes.
>
> 关于SWF的更多信息，请访问Spring Web Flow的官网。

Spring的web模块支持许多web相关的特性：

- 清晰的职责分离。每个角色——控制器，验证器，命令对象，表单对象，模型对象，`DispatcherServlet`，处理器映射，视图解析器，等等许多——的工作，都可以由相应的对象来完成。
- 强大、直观的框架和应用bean的配置。这种配置能力包括能够从不同的上下文中进行简单的引用，比如在web控制器中引用业务对象、验证器等。
- 强大的适配能力、非侵入性和灵活性。Spring MVC支持你定义任意的控制器方法签名，在特定的场景下你还可以添加适合的注解（比如`@RequestParam`、`@RequestHeader`、`@PathVariable`等）
- 可复用的业务代码，使你远离重复代码。你可以使用已有的业务对象作为命令对象或表单对象，而不需让它们去继承一个框架提供的什么基类。
- 可定制的数据绑定和验证。类型不匹配仅被认为是应用级别的验证错误，错误值、本地化日期、数字绑定等会被保存。你不需要再在表单对象使用全String字段，然后再手动将它们转换成业务对象。
- 可定制的处理器映射和视图解析。处理器映射和视图解析策略从简单的基于URL配置，到精细专用的解析策略，Spring全都支持。在这一点上，Spring比一些依赖于特定技术的web框架要更加灵活。
- 灵活的模型传递。Spring使用一个名称/值对的Map来做模型，这使得模型很容易集成、传递给任何类型的视图技术。
- 可定制的本地化信息、时区和主题解析。支持用/不用Spring标签库的JSP技术，支持JSTL，支持无需额外配置的Velocity模板，等等。
- 一个简单但功能强大的JSP标签库，通常称为Spring标签库，它提供了诸如数据绑定、主题支持等一些特性的支持。这些定制的标签为标记（markup）你的代码提供了最大程度

的灵活性。关于标签库描述符（descriptor）的更多信息，请参考附录第42章 Spring JSP标签库

- 一个Spring 2.0开始引入的JSP表单标签库。它让你在JSP页面中编写表单简单许多。关于标签库描述符（descriptor）的更多信息，请参考附录 第43章 Spring表单的JSP标签库
- 新增生命周期仅绑定到当前HTTP请求或HTTP会话的Bean类型。严格来说，这不是Spring MVC自身的特性，而是Spring MVC使用的上下文容器 `WebApplicationContext` 所提供的特性。这些bean的scope在6.5.4 请求、会话及全局会话scope一节有详细描述。

21.1.2 允许其他MVC实现

有些项目可能更倾向于使用非Spring的MVC框架。 许多团队希望仍然使用现有的技术栈，比如JSF等，这样他们掌握的技能和工具依然能发挥作用。

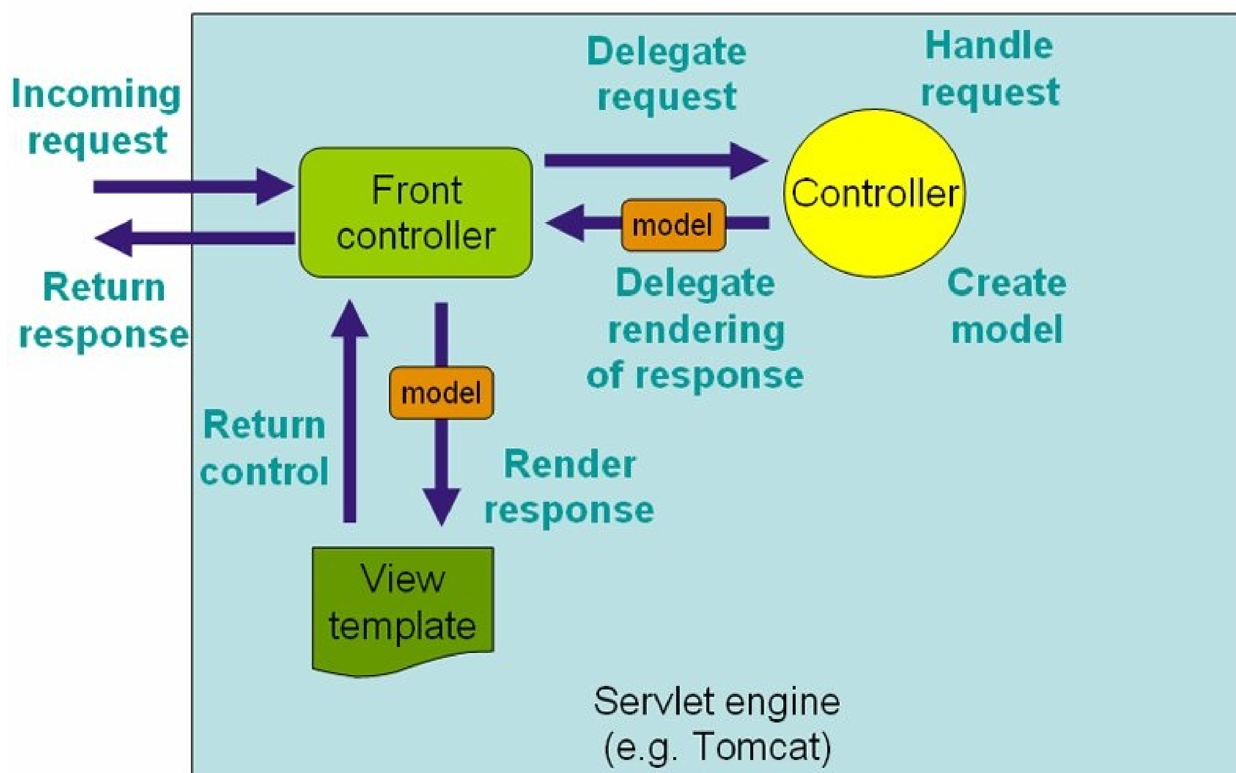
如果你确实不想使用Spring的Web MVC，但又希望能从Spring提供的一些解决方案中受益，那么将你所使用的框架和Spring进行集成也很容易。只需要在 `ContextLoaderListener` 中启动一个Spring的根应用上下文（root application context），然后你就可以在任何action对象中通过其 `ServletContext` 属性（或通过Spring对应的helper方法）取得。不需要任何侵入性的插件，因此不需要复杂的集成。从应用层的视角来看，你只是将Spring当成依赖库使用，并且将它的根应用上下文实例作为应用进入点。

即使不用Spring的Web MVC框架，你配置的其他Spring的bean和服务也都能很方便地取得。在这种场景下，Spring与其他web框架的使用不冲突。Spring只是在许多问题上提出了其他纯web MVC框架未曾提出过的解决方案，比如bean的配置、数据存取、事务处理等，仅此而已。因此，如果你只是想使用Spring的一部分特性来增强你的应用，比如Spring提供的JDBC/Hibernate事务抽象等，那么你可以将Spring作为一个中间层和/或数据存取层来使用。

21.2 DispatcherServlet

Spring MVC框架，与其他很多web的MVC框架一样：请求驱动；所有设计都围绕着一个中央Servlet来展开，它负责把所有请求分发到控制器；同时提供其他web应用开发所需要的功能。不过Spring的中央处理器，`DispatcherServlet`，能做的比这更多。它与Spring IoC容器做到了无缝集成，这意味着，Spring提供的任何特性，在Spring MVC中你都可以使用。

下图展示了Spring Web MVC的`DispatcherServlet`处理请求的工作流。熟悉设计模式的朋友会发现，`DispatcherServlet`应用的其实就是一个“前端控制器”的设计模式（其他很多优秀的web框架也都使用了这个设计模式）。



`DispatcherServlet`其实就是个`Servlet`（它继承自`HttpServlet`基类），同样也需要在你web应用的`web.xml`配置文件下声明。你需要在`web.xml`文件中把你希望`DispatcherServlet`处理的请求映射到对应的URL上去。这就是标准的Java EE Servlet配置；下面的代码就展示了对`DispatcherServlet`和路径映射的声明：

```
<web-app>
    <servlet>
        <servlet-name>example</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class
>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>example</servlet-name>
        <url-pattern>/example/*</url-pattern>
    </servlet-mapping>
</web-app>
```

> In the preceding example, all requests starting with `/example` will be handled by the `DispatcherServlet` instance named example. In a Servlet 3.0+ environment, you also have the option of configuring the Servlet container programmatically. Below is the code based equivalent of the above web.xml example:

在上面的例子中，所有路径以 `/example` 开头的请求都会被名字为 `example` 的 `DispatcherServlet` 处理。在Servlet 3.0+的环境下，你还可以用编程的方式配置Servlet容器。下面是一段这种基于代码配置的例子，它与上面定义的 `web.xml` 配置文件是等效的。

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {

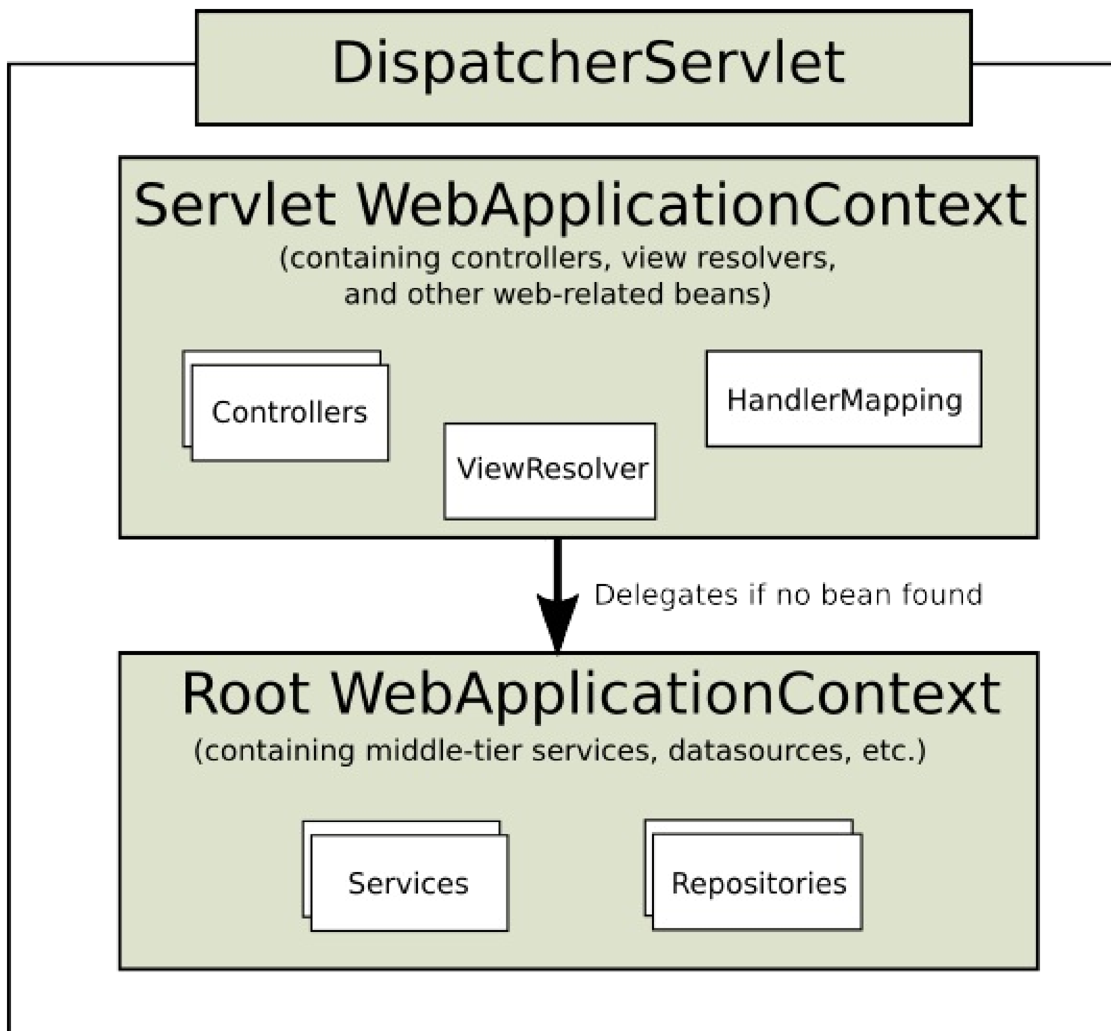
    @Override
    public void onStartup(ServletContext container) {
        ServletRegistration.Dynamic registration = container.addServlet("dispatcher",
new DispatcherServlet());
        registration.setLoadOnStartup(1);
        registration.addMapping("/example/*");
    }

}
```

`WebApplicationInitializer` 是Spring MVC提供的一个接口，它会查找你所有基于代码的配置，并应用它们来初始化Servlet 3版本以上的web容器。它有一个抽象的实现 `AbstractDispatcherServletInitializer` ，用以简化 `DispatcherServlet` 的注册工作：你只需要指定其servlet映射（mapping）即可。若想了解更多细节，可以参考基于代码的Servlet容器初始化一节。

上面只是配置Spring Web MVC的第一步，接下来你需要配置其他的一些bean（除了 `DispatcherServlet` 以外的其他bean），它们也会被Spring Web MVC框架使用到。

在6.15 应用上下文ApplicationContext的其他作用)一节中我们聊到，Spring中的 `ApplicationContext` 实例是可以有范围（scope）的。在Spring MVC中，每个 `DispatcherServlet` 都持有一个自己的上下文对象 `WebApplicationContext` ，它又继承了根（root） `WebApplicationContext` 对象中已经定义的所有bean。这些继承的bean可以在具体的Servlet实例中被重载，在每个Servlet实例中你也可以定义其scope下的新bean。



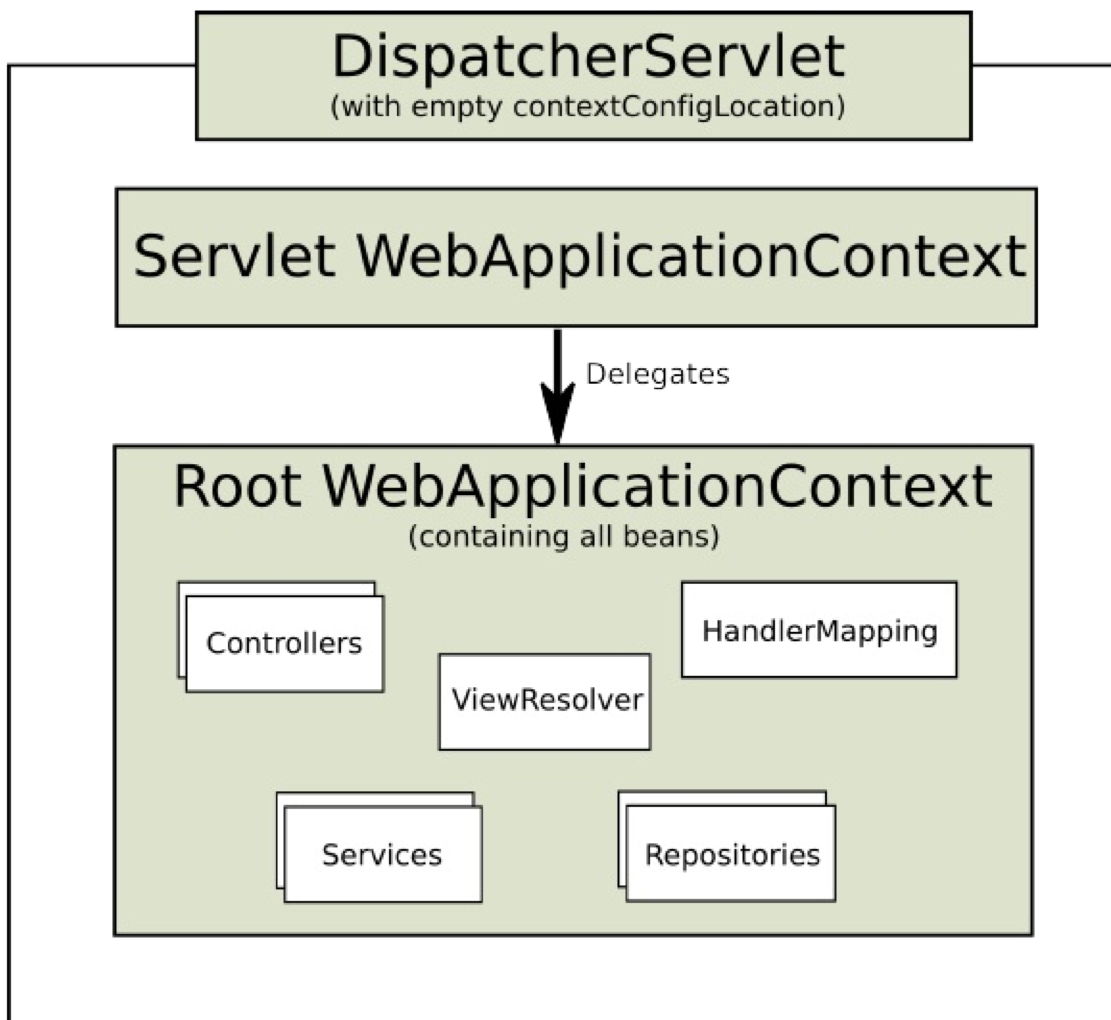
`DispatcherServlet` 的初始化过程中，Spring MVC会在你web应用的 `WEB-INF` 目录下查找一个名为[servlet-name]-servlet.xml的配置文件，并创建其中所定义的bean。如果在全局上下文中存在相同名字的bean，则它们将被新定义的同名bean覆盖。

看看下面这个 `DispatcherServlet` 的Servlet配置（定义于web.xml文件中）：

```
<web-app>
    <servlet>
        <servlet-name>golfing</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class
>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>golfing</servlet-name>
        <url-pattern>/golfing/*</url-pattern>
    </servlet-mapping>
</web-app>
```

有了以上的Servlet配置文件，你还需要在应用中的 `/WEB-INF/` 路径下创建一个 `golfing-servlet.xml` 文件，在该文件中定义所有Spring MVC相关的组件（比如bean等）。你可以通过servlet初始化参数为这个配置文件指定其他的路径（更多细节请参考下文）。

当你的应用中只需要一个 `DispatcherServlet` 时，只配置一个根context对象也是可行的。



要配置一个唯一的根context对象，可以通过在servlet初始化参数中配置一个空的contextConfigLocation来做到，如下所示：

```
<web-app>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/root-context.xml</param-value>
    </context-param>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class
>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value></param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listene
r-class>
    </listener>
</web-app>
```

`WebApplicationContext` 继承自 `ApplicationContext` ，它提供了一些web应用经常需要用到的特性。它与普通的 `ApplicationContext` 不同的地方在于，它支持主题的解析（详见21.9 主题Themes一小节），并且知道它关联到的是哪个servlet（它持有一个该 `ServletContext` 的引用）。 `WebApplicationContext` 被绑定在 `ServletContext` 中。如果需要获取它，你可以通过 `RequestContextUtils` 工具类中的静态方法来拿到这个web应用的上下文 `WebApplicationContext` 。

21.2.1 WebApplicationContext中特殊的bean类型

Spring的 `DispatcherServlet` 使用了特殊的bean来处理请求、渲染视图等，这些特定的bean是Spring MVC框架的一部分。如果你想指定使用哪个特定的bean，你可以在web应用上下文 `WebApplicationContext` 中简单地配置它们。当然这只是可选的，Spring MVC维护了一个默认的bean列表，如果你没有进行特别的配置，框架将会使用默认的bean。下一小节会介绍更多的细节，这里，我们将先快速地看一下， `DispatcherServlet` 都依赖于哪些特殊的bean来进行它的初始化。

bean的类型	作用
`HandlerMapping`	处理器映射。它会根据某些规则将进入容器的请求映射到具体的处理器以及一系列前处理器和后处理器（即处理器拦截器）上。具体的规则视 `HandlerMapping` 类的实现不同而有所不同。其最常用的一个实现支持你在控制器上添加注解，配置请求路径。当然，也存在其他的实现。
`HandlerAdapter`	处理器适配器。拿到请求所对应的处理器后，适配器将负责去调用该处理器，这使得 `DispatcherServlet` 无需关心具体的调用细节。比方说，要调用的是一个基于注解配置的控制器，那么调用前还需要从许多注解中解析出一些相应的信息。因此， `HandlerAdapter` 的主要任务就是对 `DispatcherServlet` 屏蔽这些具体的细节。
`HandlerExceptionResolver`	处理器异常解析器。它负责将捕获的异常映射到不同的视图上去，此外还支持更复杂的异常处理代码。
`ViewResolver`	视图解析器。它负责将一个代表逻辑视图名的字符串（String）映射到实际的视图类型 `View` 上。
`LocaleResolver` & `LocaleContextResolver`	地区解析器 和 地区上下文解析器。它们负责解析客户端所在的地区信息甚至时区信息，为国际化的视图定制提供了支持。
`ThemeResolver`	主题解析器。它负责解析你web应用中可用的主题，比如，提供一些个性化定制的布局等。
`MultipartResolver`	解析multi-part的传输请求，比如支持通过HTML表单进行的文件上传等。
`FlashMapManager`	FlashMap管理器。它能够存储并取回两次请求之间的 `FlashMap` 对象。后者可用于在请求之间传递数据，通常是在请求重定向的情境下使用。

21.2.2 默认的DispatcherServlet配置

上一小节讲到，`DispatcherServlet` 维护了一个列表，其中保存了其所依赖的所有bean的默认实现。这个列表保存在包 `org.springframework.web.servlet` 下的 `DispatcherServlet.properties` 文件中。

这些特殊的bean都有一些基本的默认行为。或早或晚，你可能需要对它们提供的一些默认配置进行定制。比如说，通常你需要配置 `InternalResourceViewResolver` 类提供的 `prefix` 属性，使其指向视图文件所在的目录。 这里需要理解的一个事情是，一旦你在web应用上下文 `WebApplicationContext` 中配置了某个特殊bean以后（比如 `InternalResourceViewResolver` ），实际上你也覆写了该bean的默认实现。比方说，如果你配置了 `InternalResourceViewResolver` ，那么框架就不会再使用bean `ViewResolver` 的默认实现。

在21.16节 Spring MVC的配置中，我们介绍了其他配置Spring MVC的方式，比如通过Java编程配置或者通过MVC XML命名空间进行配置。它们为配置一个Spring MVC应用提供了简易的开始方式，也不需要你对框架实现细节有太多了解。当然，无论你选用何种方式开始配置，本节所介绍的一些概念都是基础且普适的，它们对你后续的学习都应有所助益。

21.2.3 DispatcherServlet的处理流程

配置好 `DispatcherServlet` 以后，开始有请求会经过这个 `DispatcherServlet` 。此时， `DispatcherServlet` 会依照以下的次序对请求进行处理：

- 首先，搜索应用的上下文对象 `WebApplicationContext` 并把它作为一个属性（attribute）绑定到该请求上，以便控制器和其他组件能够使用它。属性的键名默认为 `DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE`
- 将地区（locale）解析器绑定到请求上，以便其他组件在处理请求（渲染视图、准备数据等）时可以获取区域相关的信息。如果你的应用不需要解析区域相关的信息，忽略它即可
- 将主题（theme）解析器绑定到请求上，以便其他组件（比如视图等）能够了解要渲染哪个主题文件。同样，如果你不需要使用主题相关的特性，忽略它即可
- 如果你配置了multipart文件处理器，那么框架将查找该文件是不是multipart（分为多个部分连续上传）的。若是，则将该请求包装成一个 `MultipartHttpServletRequest` 对象，以便处理链中的其他组件对它做进一步的处理。关于Spring对multipart文件传输处理的支持，读者可以参考21.10 Spring的multipart（文件上传）支持一小节
- 为该请求查找一个合适的处理器。如果可以找到对应的处理器，则与该处理器关联的整条执行链（前处理器、后处理器、控制器等）都会被执行，以完成相应模型的准备或视图的渲染
- 如果处理器返回的是一个模型（model），那么框架将渲染相应的视图。若没有返回任何模型（可能是因为前后的处理器出于某些原因拦截了请求等，比如，安全问题），则框架不会渲染任何视图，此时认为对请求的处理可能已经由处理链完成了

如果在处理请求的过程中抛出了异常，那么上下文 `WebApplicationContext` 对象中所定义的异常处理器将会负责捕获这些异常。通过配置你自己的异常处理器，你可以定制自己处理异常的方式。

Spring的 `DispatcherServlet` 也允许处理器返回一个Servlet API规范中定义的 最后修改时间戳（*last-modification-date*） 值。决定请求最后修改时间的方式很直接： `DispatcherServlet` 会先查找合适的处理器映射来找到请求对应的处理器，然后检测它是否实现了 *LastModified* 接口。若是，则调用接口的 `long getLastModified(request)` 方法，并将该返回值返回给客户端。

你可以定制 `DispatcherServlet` 的配置，具体的做法，是在 `web.xml` 文件中，Servlet的声明元素上添加一些Servlet的初始化参数（通过 `init-param` 元素）。该元素可选的参数列表如下：

可选参数	解释
`contextClass`	任意实现了 `WebApplicationContext` 接口的类。这个类会初始化该servlet所需要用到的上下文对象。默认情况下，框架会使用一个 `XmlWebApplicationContext` 对象。
`contextConfigLocation`	一个指定了上下文配置文件路径的字符串，该值会被传入给 `contextClass` 所指定的上下文实例对象。该字符串内可以包含多个字符串，字符串之间以逗号分隔，以此支持你进行多个上下文的配置。在多个上下文中重复定义的bean，以最后加载的bean定义为准
`namespace`	`WebApplicationContext` 的命名空间。默认是 `[servlet-name]-servlet`

21.3 控制器(Controller)的实现

> ...Spring implements a controller in a very abstract way, which enables you to create a wide variety of controllers.

控制器作为应用程序逻辑的处理入口，它会负责去调用你已经实现的一些服务。通常，一个控制器会接收并解析用户的请求，然后把它转换成一个模型交给视图，由视图渲染出页面最终呈现给用户。Spring对控制器的定义非常宽松，这意味着你在实现控制器时非常自由。

Spring 2.5以后引入了基于注解的编程模型，你可以在你的控制器实现上添加 `@RequestMapping` 、 `@RequestParam` 、 `@ModelAttribute` 等注解。注解特性既支持基于Servlet的MVC，也可支持基于Portlet的MVC。通过此种方式实现的控制器既无需继承某个特定的基类，也无需实现某些特定的接口。而且，它通常也不会直接依赖于Servlet或Portlet的API来进行编程，不过你仍然可以很容易地获取Servlet或Portlet相关的变量、特性和设施等。

> 在Spring项目的官方Github上你可以找到许多项目，它们对本节所述以后的注解支持提供了进一步增强，比如说MvcShowcase，MvcAjax，MvcBasic，PetClinic，PetCare等。

```
@Controller
public class HelloWorldController {

    @RequestMapping("/helloWorld")
    public String helloWorld(Model model) {
        model.addAttribute("message", "Hello World!");
        return "helloWorld";
    }
}
```

你可以看到， `@Controller` 注解和 `@RequestMapping` 注解支持多样的方法名和方法签名。在上面这个例子中，方法接受一个 `Model` 类型的参数并返回一个字符串 `String` 类型的视图名。但事实上，方法所支持的参数和返回值有非常多的选择，这个我们在本小节的后面部分会提及。 `@Controller` 和 `@RequestMapping` 及其他的一些注解，共同构成了Spring MVC框架的基本实现。本节将详细地介绍这些注解，以及它们在一个Servlet环境下最常被使用到的一些场景。

21.3.1 使用@Controller注解定义一个控制器

> [Original] The `@Controller` annotation indicates that a particular class serves the role of a controller. Spring does not require you to extend any controller base class or reference the Servlet API. However, you can still reference Servlet-specific features if you need to.

`@Controller` 注解表明了一个类是作为控制器的角色而存在的。Spring不要求你去继承任何控制器基类，也不要求你去实现Servlet的那套API。当然，如果你需要的话也可以去使用任何与Servlet相关的特性和设施。

> [Original] The `@Controller` annotation acts as a stereotype for the annotated class, indicating its role. The dispatcher scans such annotated classes for mapped methods and detects `@RequestMapping` annotations (see the next section).

`@Controller` 注解可以认为是被标注类的原型（stereotype），表明了这个类所承担的角色。分派器（ `DispatcherServlet` ）会扫描所有注解了 `@Controller` 的类，检测其中通过 `@RequestMapping` 注解配置的方法（详见下一小节）。

> [Original] You can define annotated controller beans explicitly, using a standard Spring bean definition in the dispatcher's context. However, the `@Controller` stereotype also allows for autodetection, aligned with Spring general support for detecting component classes in the classpath and auto-registering bean definitions for them.

当然，你也可以不使用 `@Controller` 注解而显式地去定义被注解的bean，这点通过标准的Spring bean的定义方式，在dispather的上下文属性下配置即可做到。但是 `@Controller` 原型是可以被框架自动检测的，Spring支持classpath路径下组件类的自动检测，以及对已定义bean的自动注册。

> [Original] To enable autodetection of such annotated controllers, you add component scanning to your configuration. Use the spring-context schema as shown in the following XML snippet:

你需要在配置中加入组件扫描的配置代码来开启框架对注解控制器的自动检测。请使用下面XML代码所示的spring-context schema：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.springframework.samples.petclinic.web"/>

    <!-- ... -->

</beans>
```

21.3.2 使用@RequestMapping注解映射请求路径

你可以使用 `@RequestMapping` 注解来将请求URL，如 `/appointments` 等，映射到整个类上或某个特定的处理器方法上。一般来说，类级别的注解负责将一个特定（或符合某种模式）的请求路径映射到一个控制器上，同时通过方法级别的注解来细化映射，即根据特定的HTTP请求方法（“GET”“POST”方法等）、HTTP请求中是否携带特定参数等条件，将请求映射到匹配的方法上。

下面这段代码示例来自Petcare，它展示了在Spring MVC中如何在控制器上使用 `@RequestMapping` 注解：

```
@Controller
@RequestMapping("/appointments")
public class AppointmentsController {

    private final AppointmentBook appointmentBook;

    @Autowired
    public AppointmentsController(AppointmentBook appointmentBook) {
        this.appointmentBook = appointmentBook;
    }

    @RequestMapping(method = RequestMethod.GET)
    public Map<String, Appointment> get() {
        return appointmentBook.getAppointmentsForToday();
    }

    @RequestMapping(path = "/{day}", method = RequestMethod.GET)
    public Map<String, Appointment> getForDay(@PathVariable @DateTimeFormat(iso=ISO.DA
TE) Date day, Model model) {
        return appointmentBook.getAppointmentsForDay(day);
    }

    @RequestMapping(path = "/new", method = RequestMethod.GET)
    public AppointmentForm getNewForm() {
        return new AppointmentForm();
    }

    @RequestMapping(method = RequestMethod.POST)
    public String add(@Valid AppointmentForm appointment, BindingResult result) {
        if (result.hasErrors()) {
            return "appointments/new";
        }
        appointmentBook.addAppointment(appointment);
        return "redirect:/appointments";
    }
}
```

在上面的示例中，许多地方都使用到了 `@RequestMapping` 注解。第一次使用点是作用于类级别的，它指示了所有 `/appointments` 开头的路径都会被映射到控制器下。 `get()` 方法上的 `@RequestMapping` 注解对请求路径进行了进一步细化：它仅接受GET方法的请求。这样，一个请求路径为 `/appointments` 、HTTP方法为GET的请求，将会最终进入到这个方法被处理。 `add()` 方法也做了类似的细化，而 `getNewForm()` 方法则同时注解了能够接受的请求的HTTP方法和路径。这种情况下，一个路径为 `appointments/new` 、HTTP方法为GET的请求将会被这个方法所处理。

`getForDay()` 方法则展示了使用 `@RequestMapping` 注解的另一个技巧：URI模板。（关于URI模板，请见下小节）

类级别的 `@RequestMapping` 注解并不是必须的。不配置的话则所有的路径都是绝对路径，而非相对路径。以下的代码示例来自PetClinic，它展示了一个具有多个处理器方法的控制器：

```
@Controller
public class ClinicController {

    private final Clinic clinic;

    @Autowired
    public ClinicController(Clinic clinic) {
        this.clinic = clinic;
    }

    @RequestMapping("/")
    public void welcomeHandler() {
    }

    @RequestMapping("/vets")
    public ModelMap vetsHandler() {
        return new ModelMap(this.clinic.getVets());
    }
}
```

以上代码没有指定请求必须是GET方法还是PUT/POST或其他方法， `@RequestMapping` 注解默认会映射所有的HTTP请求方法。如果仅想接收某种请求方法，请在注解中指定之 `@RequestMapping(method=GET)` 以缩小范围。

@Controller和面向切面（AOP）代理

有时，我们希望在运行时使用AOP代理来装饰控制器，比如当你直接在控制器上使用 `@Transactional` 注解时。这种情况下，我们推荐使用类级别（在控制器上使用）的代理方式。这一般是代理控制器的默认做法。如果控制器必须实现一些接口，而该接口又不支持Spring Context的回调（比如 `InitializingBean` ， `*Aware` 等接口），那要配置类级别的代理就必须手动配置了。比如，原来的配置文件 `<tx:annotation-driven/>` 需要显式配置为 `<tx:annotation-driven proxy-target-class="true"/>` 。

Spring MVC 3.1中新增支持@RequestMapping的一些类

> They are recommended for use and even required to take advantage of new features in Spring MVC 3.1 and going forward.

Spring 3.1中新增了一组类用以增强 `@RequestMapping` ，分别是 `RequestMappingHandlerMapping` 和 `RequestMappingHandlerAdapter` 。我们推荐你用一用。有部分Spring MVC 3.1之后新增的特性，这两个注解甚至是必须的。在MVC命名空间和MVC Java编程配置方式下，这组类及其新特性默认是开启的。但若你使用其他配置方式，则该特性必须手动配置才能使用。本小节将简要介绍一下，新类相比之前的一些重要变化。

在Spring 3.1之前，框架会在两个不同的阶段分别检查类级别和方法级别的请求映射——首先， `DefaultAnnotationHanlderMapping` 会先在类级别上选中一个控制器，然后再通过 `AnnotationMethodHandlerAdapter` 定位到具体要调用的方法。

> [Original] With the new support classes in Spring 3.1, the `RequestMappingHandlerMapping` is the only place where a decision is made about which method should process the request. Think of controller methods as a collection of unique endpoints with mappings for each method derived from type and method-level `@RequestMapping` information.

现在有了Spring 3.1后引入的这组新类， `RequestMappingHandlerMapping` 成为了这两个决策实际发生的唯一一个地方。你可以把控制器中的一系列处理方法当成是一系列独立的服务节点，每个从类级别和方法级别的 `@RequestMapping` 注解中获取到足够请求1路径映射信息。

> [Original] This enables some new possibilities. For once a `HandlerInterceptor` or a `HandlerExceptionResolver` can now expect the Object-based handler to be a `HandlerMethod`, which allows them to examine the exact method, its parameters and associated annotations. The processing for a URL no longer needs to be split across different controllers.

这种新的处理方式带来了新的可能性。之前的 `HandlerInterceptor` 或 `HandlerExceptionResolver` 现在可以确定拿到的这个处理器肯定是一个 `HandlerMethod` 类型，因此它能够精确地了解这个方法的所有信息，包括它的参数、应用于其上的注解等。这样，内部对于一个URL的处理流程再也不需要分隔到不同的控制器里面去执行了。

> [Original] There are also several things no longer possible: [Original] *Select a controller first with a `SimpleUrlHandlerMapping` or `BeanNameUrlHandlerMapping` and then narrow the method based on `@RequestMapping` annotations. [Original]* Rely on method names as a fall-back mechanism to disambiguate between two `@RequestMapping` methods that don't have an explicit path mapping URL path but otherwise match equally, e.g. by HTTP method. In the new support classes `@RequestMapping` methods have to be mapped uniquely. [Original] * Have a single default method (without an explicit path mapping) with which requests are processed if no other controller method matches more concretely. In the new support classes if a matching method is not found a 404 error is raised.

同时，也有其他的一些变化，比如有些事情就没法这么玩儿了：

- 先通过 `SimpleUrlHandlerMapping` 或 `BeanNameUrlHandlerMapping` 来拿到负责处理请求的控制器，然后通过 `@RequestMapping` 注解配置的信息来定位到具体的处理方法；
- 依靠方法名称来作为选择处理方法的标准。比如说，两个注解了 `@RequestMapping` 的方法除了方法名称拥有完全相同的URL映射和HTTP请求方法。在新版本下， `@RequestMapping` 注解的方法必须具有唯一的请求映射；
- 定义一个默认方法（即没有声明路径映射），在请求路径无法被映射到控制器下更精确的方法上去时，为该请求提供默认处理。在新版本中，如果无法为一个请求找到合适的处理方法，那么一个404错误将被抛出；

> [Original] The above features are still supported with the existing support classes. However to take advantage of new Spring MVC 3.1 features you'll need to use the new support classes.

如果使用原来的类，以上的功能还是可以做到。但是，如果要享受Spring MVC 3.1版本带来的方便特性，你就需要去使用新的类。

> [Original] ## URI Template Patterns

URI模板

> [Original] URI templates can be used for convenient access to selected parts of a URL in a `@RequestMapping` method.

URI模板可以为快速访问 `@RequestMapping` 中指定的URL的一个特定的部分提供很大的便利。

> [Original] A URI Template is a URI-like string, containing one or more variable names. When you substitute values for these variables, the template becomes a URI. The proposed RFC for URI Templates defines how a URI is parameterized. For example, the URI Template `http://www.example.com/users/{userId}` contains the variable userId. Assigning the value fred to the variable yields `http://www.example.com/users/fred`.

URI模板是一个类似于URI的字符串，只不过其中包含了一个或多个的变量名。当你使用实际的值去填充这些变量名的时候，模板就退化成了一个URI。在URI模板的RFC提议中定义了一个URI是如何进行参数化的。比如说，一个这个URI模板 `http://www.example.com/users/{userId}` 就包含了一个变量名*userId*。将值*fred*赋给这个变量名后，它就变成了一个URI： `http://www.example.com/users/fred` 。

> [Original] In Spring MVC you can use the `@PathVariable` annotation on a method argument to bind it to the value of a URI template variable:

在Spring MVC中你可以在方法参数上使用 `@PathVariable` 注解，将其与URI模板中的参数绑定起来：

```
@RequestMapping(path="/owners/{ownerId}", method=RequestMethod.GET)
public String findOwner(@PathVariable String ownerId, Model model) {
    Owner owner = ownerService.findOwner(ownerId);
    model.addAttribute("owner", owner);
    return "displayOwner";
}
```

> [Original] The URI Template " `/owners/{ownerId}` " specifies the variable name `ownerId` . When the controller handles this request, the value of `ownerId` is set to the value found in the appropriate part of the URI. For example, when a request comes in for `/owners/fred` , the value of `ownerId` is `fred` .

URI模板" `/owners/{ownerId}` "指定了一个变量，名为 `ownerId` 。当控制器处理这个请求的时候， `ownerId` 的值就会被URI模板中对应部分的值所填充。比如说，如果请求的URI是 `/owners/fred` ，此时变量 `ownerId` 的值就是 `fred` .`

> 为了处理 `@PathVariables` 注解，Spring MVC必须通过变量名来找到URI模板中相对应的变量。你可以在注解中直接声明：
>
> ```
> @RequestMapping(path="/owners/{ownerId}}", method=RequestMethod.GET)
> public String findOwner(@PathVariable("ownerId") String theOwner, Model model) {
>     // 具体的方法代码…
> }
> ```
>
> 或者，如果URI模板中的变量名与方法的参数名是相同的，则你可以不必再指定一次。只要你在编译的时候留下debug信息，Spring MVC就可以自动匹配URL模板中与方法参数名相同的变量名。
>
> ```
> @RequestMapping(path="/owners/{ownerId}", method=RequestMethod.GET)
> public String findOwner(@PathVariable String ownerId, Model model) {
>     // 具体的方法代码…
> }
> ```
>
> [Original] A method can have any number of `@PathVariable` annotations:

一个方法可以拥有任意数量的 `@PathVariable` 注解：

```
@RequestMapping(path="/owners/{ownerId}/pets/{petId}", method=RequestMethod.GET)
public String findPet(@PathVariable String ownerId, @PathVariable String petId, Model
model) {
    Owner owner = ownerService.findOwner(ownerId);
    Pet pet = owner.getPet(petId);
    model.addAttribute("pet", pet);
    return "displayPet";
}
```

> [Original] When a `@PathVariable` annotation is used on a `Map<String, String>` argument, the map is populated with all URI template variables.

当 `@PathVariable` 注解被应用于 `Map<String, String>` 类型的参数上时，框架会使用所有URI模板变量来填充这个map。

> [Original] A URI template can be assembled from type and path level *@RequestMapping* annotations. As a result the `findPet()` method can be invoked with a URL such as `/owners/42/pets/21`.

URI模板可以从类级别和方法级别的 *@RequestMapping* 注解获取数据。因此，像这样的 `findPet()` 方法可以被类似于 `/owners/42/pets/21` 这样的URL路由并调用到：

```
_@Controller_
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

    @RequestMapping("/pets/{petId}")
    public void findPet(_@PathVariable_ String ownerId, _@PathVariable_ String petId,
Model model) {
        // 方法实现体这里忽略
    }

}
```

> [Original] A `@PathVariable` argument can be of *any simple type* such as int, long, Date, etc. Spring automatically converts to the appropriate type or throws a `TypeMismatchException` if it fails to do so. You can also register support for parsing additional data types. See the section called "Method Parameters And Type Conversion" and the section called "Customizing WebDataBinder initialization".

`@PathVariable` 可以被应用于所有 简单类型 的参数上，比如int、long、Date等类型。Spring会自动地帮你把参数转化成合适的类型，如果转换失败，就抛出一个 `TypeMismatchException` 。如果你需要处理其他数据类型的转换，也可以注册自己的类。若需要更详细的信息可以参考“方法参数与类型转换”一节和“定制WebDataBinder初始化过程”一节

带正则表达式的URI模板

> [Original] Sometimes you need more precision in defining URI template variables. Consider the URL `"/spring-web/spring-web-3.0.5.jar"`. How do you break it down into multiple parts?

有时候你可能需要更准确地描述一个URI模板的变量，比如说这个URL：`"/spring-web/spring-web-3.0.5.jar`。你要怎么把它分解成几个有意义的部分呢？

> [Original] The `@RequestMapping` annotation supports the use of regular expressions in URI template variables. The syntax is `{varName:regex}` where the first part defines the variable name and the second - the regular expression.For example:

`@RequestMapping` 注解支持你在URI模板变量中使用正则表达式。语法是 `{varName:regex}`，其中第一部分定义了变量名，第二部分就是你所要应用的正则表达式。比如下面的代码样例：

```
@RequestMapping("/spring-web/{symbolicName:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{extension:\\.[a-z]+}")
    public void handle(@PathVariable String version, @PathVariable String extension) {
        // 代码部分省略...
    }
}
```

Path Patterns（不好翻，容易掉韵味）

> [Original] In addition to URI templates, the `@RequestMapping` annotation also supports Ant-style path patterns (for example, `/myPath/*.do`). A combination of URI template variables and Ant-style globs is also supported (e.g. `/owners/*/pets/{petId}`).

除了URI模板外，`@RequestMapping` 注解还支持Ant风格的路径模式（如 `/myPath/*.do` 等）。不仅如此，还可以把URI模板变量和Ant风格的glob组合起来使用（比如 `/owners/*/pets/{petId}` 这样的用法等）。

路径样式的匹配(Path Pattern Comparison)

> [Original] When a URL matches multiple patterns, a sort is used to find the most specific match.

当一个URL同时匹配多个模板（pattern）时，我们将需要一个算法来决定其中最匹配的一个。

> [Original] A pattern with a lower count of URI variables and wild cards is considered more specific. For example `/hotels/{hotel}/*` has 1 URI variable and 1 wild card and is considered more specific than `/hotels/{hotel}/**` which as 1 URI variable and 2 wild cards.

URI模板变量的数目和通配符数量的总和最少的那个路径模板更准确。举个例子，`/hotels/{hotel}/*` 这个路径拥有一个URI变量和一个通配符，而 `/hotels/{hotel}/**` 这个路径则拥有一个URI变量和两个通配符，因此，我们认为前者是更准确的路径模板。

> [Original] If two patterns have the same count, the one that is longer is considered more specific. For example `/foo/bar*` is longer and considered more specific than `/foo/*`.

如果两个模板的URI模板数量和通配符数量总和一致，则路径更长的那个模板更准确。举个例子，`/foo/bar*` 就被认为比 `/foo/*` 更准确，因为前者的路径更长。

> [Original] When two patterns have the same count and length, the pattern with fewer wild cards is considered more specific. For example `/hotels/{hotel}` is more specific than `/hotels/*`.

如果两个模板的数量和长度均一致，则那个具有更少通配符的模板是更加准确的。比如，`/hotels/{hotel}` 就比 `/hotels/*` 更精确。

> [Original] There are also some additional special rules:

除此之外，还有一些其他的规则：

> [Original] *The **default mapping pattern** `/*` `is less specific than any other pattern. For example` /api/{a}/{b}/{c}` is more specific.
>
> [Original] *A **prefix pattern** such as `/public/*` `is less specific than any other pattern that doesn't contain double wildcards. For example` /public/path3/{a}/{b}/{c}` is more specific.

- 默认的通配模式 `/**` 比其他所有的模式都更“不准确”。比方说，`/api/{a}/{b}/{c}` 就比默认的通配模式 `/**` 要更准确
- 前缀通配（比如 `/public/**`)被认为比其他任何不包括双通配符的模式更不准确。比如说，`/public/path3/{a}/{b}/{c}` 就比 `/public/**` 更准确

> [Original] For the full details see `AntPatternComparator` in `AntPathMatcher`. Note that the PathMatcher can be customized (see Section 21.16.11, "Path Matching" in the section on configuring Spring MVC).

更多的细节请参考这两个类：`AntPatternComparator` 和 `AntPathMatcher` 。值得一提的是，PathMatcher类是可以配置的（见“配置Spring MVC”一节中的21.16.11 路径的匹配一节)。

带占位符的路径模式（path patterns）

> [Original] Patterns in `@RequestMapping` annotations support ${…} placeholders against local properties and/or system properties and environment variables. This may be useful in cases where the path a controller is mapped to may need to be customized through configuration. For more information on placeholders, see the javadocs of the `PropertyPlaceholderConfigurer` class.

`@RequestMapping` 注解支持在路径中使用占位符，以取得一些本地配置、系统配置、环境变量等。这个特性有时很有用，比如说控制器的映射路径需要通过配置来定制的场景。如果想了解更多关于占位符的信息，可以参考 `PropertyPlaceholderConfigurer` 这个类的文档。

Suffix Pattern Matching

后缀模式匹配

> [Original] By default Spring MVC performs `".*"` suffix pattern matching so that a controller mapped to `/person` is also implicitly mapped to `/person.*`. This makes it easy to request different representations of a resource through the URL path (e.g. `/person.pdf`, `/person.xml`).

Spring MVC默认采用 `".*"` 的后缀模式匹配来进行路径匹配，因此，一个映射到 `/person` 路径的控制器也会隐式地被映射到 `/person.*` 。这使得通过URL来请求同一资源文件的不同格式变得更简单（比如 `/person.pdf` ， `/person.xml` ）。

> [Original] Suffix pattern matching can be turned off or restricted to a set of path extensions explicitly registered for content negotiation purposes. This is generally recommended to minimize ambiguity with common request mappings such as `/person/{id}` where a dot might not represent a file extension, e.g. `/person/joe@email.com` vs `/person/joe@email.com.json)`. Furthermore as explained in the note below suffix pattern matching as well as content negotiation may be used in some circumstances to attempt malicious attacks and there are good reasons to restrict them meaningfully.

你可以关闭默认的后缀模式匹配，或者显式地将路径后缀限定到一些特定格式上for content negotiation purpose。我们推荐这样做，这样可以减少映射请求时可以带来的一些二义性，比如请求以下路径 `/person/{id}` 时，路径中的点号后面带的可能不是描述内容格式，比如 `/person/joe@email.com` vs `/person/joe@email.com.json` 。而且正如下面马上要提到的，后缀模式通配以及内容协商有时可能会被黑客用来进行攻击，因此，对后缀通配进行有意义的限定是有好处的。

> [Original] See Section 21.16.11, "Path Matching" for suffix pattern matching configuration and also Section 21.16.6, "Content Negotiation" for content negotiation configuration.

关于后缀模式匹配的配置问题，可以参考第21.16.11小节 "路径匹配"；关于内容协商的配置问题，可以参考第21.16.6小节 "内容协商"的内容。

后缀模式匹配与RFD

> [Original] Reflected file download (RFD) attack was first described in a paper by Trustwave in 2014. The attack is similar to XSS in that it relies on input (e.g. query parameter, URI variable) being reflected in the response. However instead of inserting JavaScript into HTML, an RFD attack relies on the browser switching to perform a download and treating the response as an executable script if double-clicked based on the file extension (e.g. .bat, .cmd).

RFD(Reflected file download)攻击最先是2014年在Trustwave的一篇论文中被提出的。它与XSS攻击有些相似，因为这种攻击方式也依赖于某些特征，即需要你的输入（比如查询参数，URI变量等）等也在输出（response）中以某种形式出现。不同的是，RFD攻击并不是通过在HTML中写入JavaScript代码进行，而是依赖于浏览器来跳转到下载页面，并把特定格式（比如.bat，.cmd等）的response当成是可执行脚本，双击它就会执行。

> [Original] In Spring MVC `@ResponseBody` and `ResponseEntity` methods are at risk because they can render different content types which clients can request including via URL path extensions. Note however that neither disabling suffix pattern matching nor disabling the use of path extensions for content negotiation purposes alone are effective at preventing RFD attacks.

Spring MVC的 `@ResponseBody` 和 `ResponseEntity` 方法是有风险的，因为它们会根据客户的请求——包括URL的路径后缀，来渲染不同的内容类型。因此，禁用后缀模式匹配或者禁用仅为内容协商开启的路径文件后缀名携带，都是防范RFD攻击的有效方式。

> [Original] For comprehensive protection against RFD, prior to rendering the response body Spring MVC adds a `Content-Disposition:inline;filename=f.txt` header to suggest a fixed and safe download file filename. This is done only if the URL path contains a file extension that is neither whitelisted nor explicitly registered for content negotiation purposes. However it may potentially have side effects when URLs are typed directly into a browser.

若要开启对RFD更高级的保护模式，可以在Spring MVC渲染开始请求正文之前，在请求头中增加一行配置 `Content-Disposition:inline;filename=f.txt` ，指定固定的下载文件的文件名。这仅在URL路径中包含了一个文件符合以下特征的拓展名时适用：该扩展名既不在信任列表

（白名单）中，也没有被显式地被注册于内容协商时使用。并且这种做法还可以有一些副作用，比如，当URL是通过浏览器手动输入的时候。

> [Original] Many common path extensions are whitelisted by default. Furthermore REST API calls are typically not meant to be used as URLs directly in browsers. Nevertheless applications that use custom `HttpMessageConverter` implementations can explicitly register file extensions for content negotiation and the Content-Disposition header will not be added for such extensions. See Section 21.16.6, "Content Negotiation".

很多常用的路径文件后缀默认是被信任的。另外，REST的API一般是不应该直接用做URL的。不过，你可以自己定制 `HttpMessageConverter` 的实现，然后显式地注册用于内容协商的文件类型，这种情形下Content-Disposition头将不会被加入到请求头中。详见第21.16.6节中"内容协商"的内容。

> [Original] This was originally introduced as part of work for CVE-2015-5211. Below are additional recommendations from the report:
>
> - Encode rather than escape JSON responses. This is also an OWASP XSS recommendation. For an example of how to do that with Spring see spring-jackson-owasp.
> - Configure suffix pattern matching to be turned off or restricted to explicitly registered suffixes only.
> - Configure content negotiation with the properties "useJaf" and "ignoreUnknownPathExtensions" set to false which would result in a 406 response for URLs with unknown extensions. Note however that this may not be an option if URLs are naturally expected to have a dot towards the end.
> - Add `X-Content-Type-Options: nosniff` header to responses. Spring Security 4 does this by default.

感觉这节的翻译质量还有限，需要继续了解XSS攻击和RFD攻击的细节再翻。

矩阵变量

> [Original] The URI specification RFC 3986 defines the possibility of including name-value pairs within path segments. There is no specific term used in the spec. The general "URI path parameters" could be applied although the more unique "Matrix URIs", originating from an old post by Tim Berners-Lee, is also frequently used and fairly well known. Within Spring MVC these are referred to as matrix variables.

原来的URI规范RFC 3986中允许在路径段落中携带键值对，但规范没有明确给这样的键值对定义术语。有人叫"URI路径参数"，也有叫"矩阵URI"的。后者是Tim Berners-Lee首先在其博客中提到的术语，被使用得要更加频繁一些，知名度也更高些。而在Spring MVC中，我们称

这样的键值对为矩阵变量。

> [Original] Matrix variables can appear in any path segment, each matrix variable separated with a ";" (semicolon). For example: `"/cars;color=red;year=2012"`. Multiple values may be either "," (comma) separated `"color=red,green,blue"` or the variable name may be repeated `"color=red;color=green;color=blue"`.

矩阵变量可以在任何路径段落中出现，每对矩阵变量之间使用一个分号“;”隔开。比如这样的URI：`"/cars;color=red;year=2012"`。多个值可以用逗号隔开`"color=red,green,blue"`，或者重复变量名多次`"color=red;color=green;color=blue"`。

> [Original] If a URL is expected to contain matrix variables, the request mapping pattern must represent them with a URI template. This ensures the request can be matched correctly regardless of whether matrix variables are present or not and in what order they are provided.

如果一个URL有可能需要包含矩阵变量，那么在请求路径的映射配置上就需要使用URI模板来体现这一点。这样才能确保请求可以被正确地映射，而不管矩阵变量在URI中是否出现、出现的次序是怎样等。

> [Original] Below is an example of extracting the matrix variable "q":

下面是一个例子，展示了我们如何从矩阵变量中获取到变量“q”的值：

```
// GET /pets/42;q=11;r=22

@RequestMapping(path = "/pets/{petId}", method = RequestMethod.GET)
public void findPet(@PathVariable String petId, @MatrixVariable int q) {

    // petId == 42
    // q == 11

}
```

> [Original] Since all path segments may contain matrix variables, in some cases you need to be more specific to identify where the variable is expected to be:

由于任意路径段落中都可以含有矩阵变量，在某些场景下，你需要用更精确的信息来指定一个矩阵变量的位置：

```
// GET /owners/42;q=11/pets/21;q=22

@RequestMapping(path = "/owners/{ownerId}/pets/{petId}", method = RequestMethod.GET)
public void findPet(
    @MatrixVariable(name="q", pathVar="ownerId") int q1,
    @MatrixVariable(name="q", pathVar="petId") int q2) {

    // q1 == 11
    // q2 == 22

}
```

> [Original] A matrix variable may be defined as optional and a default value specified:

你也可以声明一个矩阵变量不是必须出现的，并给它赋一个默认值：

```
// GET /pets/42

@RequestMapping(path = "/pets/{petId}", method = RequestMethod.GET)
public void findPet(@MatrixVariable(required=false, defaultValue="1") int q) {

    // q == 1

}
```

> [Original] All matrix variables may be obtained in a Map:

也可以通过一个Map来存储所有的矩阵变量：

```
// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

@RequestMapping(path = "/owners/{ownerId}/pets/{petId}", method = RequestMethod.GET)
public void findPet(
    @MatrixVariable Map<String, String> matrixVars,
    @MatrixVariable(pathVar="petId") Map<String, String> petMatrixVars) {

    // matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]
    // petMatrixVars: ["q" : 11, "s" : 23]

}
```

> [Original] Note that to enable the use of matrix variables, you must set the `removeSemicolonContent` property of `RequestMappingHandlerMapping` to `false`. By default it is set to `true`.

如果要允许矩阵变量的使用，你必须把 `RequestMappingHandlerMapping` 类的 `removeSemicolonContent` 属性设置为 `false` 。该值默认是 `true` 的。

> [Original] The MVC Java config and the MVC namespace both provide options for enabling the use of matrix variables.
>
> MVC的Java编程配置和命名空间配置都提供了启用矩阵变量的方式。
>
> [Original] If you are using Java config, The Advanced Customizations with MVC Java Config section describes how the `RequestMappingHandlerMapping` can be customized.
>
> 如果你是使用Java编程的方式，“MVC Java高级定制化配置”一节描述了如何对 `RequestMappingHandlerMapping` 进行定制。
>
> [Original] In the MVC namespace, the `<mvc:annotation-driven>` element has an `enable-matrix-variables` attribute that should be set to `true`. By default it is set to `false`.
>
> 而使用MVC的命名空间配置时，你可以把 `<mvc:annotation-driven>` 元素下的 `enable-matrix-variables` 属性设置为 `true`。该值默认情况下是配置为 `false` 的。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven enable-matrix-variables="true"/>

</beans>
```

可消费的媒体类型

> [Original] You can narrow the primary mapping by specifying a list of consumable media types. The request will be matched only if the *Content-Type* request header matches the specified media type. For example:

你可以指定一组可消费的媒体类型，缩小映射的范围。这样只有当请求头中 *Content-Type* 的值与指定可消费的媒体类型中有相同的时候，请求才会被匹配。比如下面这个例子：

```
@Controller
@RequestMapping(path = "/pets", method = RequestMethod.POST, consumes="application/jso
n")
public void addPet(@RequestBody Pet pet, Model model) {
    // 方法实现省略
}
```

> [Original] Consumable media type expressions can also be negated as in *!text/plain* to match to all requests other than those with *Content-Type* of *text/plain*. Also consider using constants provided in `MediaType` such as `APPLICATION_JSON_VALUE` and `APPLICATION_JSON_UTF8_VALUE`.

指定可消费媒体类型的表达式中还可以使用否定，比如，可以使用 *!text/plain* 来匹配所有请求头 *Content-Type* 中不含 *text/plain* 的请求。同时，在 `MediaType` 类中还定义了一些常量，比如 `APPLICATION_JSON_VALUE`、`APPLICATION_JSON_UTF8_VALUE` 等，推荐更多地使用它们。

> [Original] The *consumes* condition is supported on the type and on the method level. Unlike most other conditions, when used at the type level, method-level consumable types override rather than extend type-level consumable types.
>
> *consumes* 属性提供的是方法级的类型支持。与其他属性不同，当在类型级使用时，方法级的消费类型将覆盖类型级的配置，而非继承关系。

可生产的媒体类型

> [Original] You can narrow the primary mapping by specifying a list of producible media types. The request will be matched only if the *Accept* request header matches one of these values. Furthermore, use of the *produces* condition ensures the actual content type used to generate the response respects the media types specified in the *produces* condition. For example:

你可以指定一组可生产的媒体类型，缩小映射的范围。这样只有当请求头中 *Accept* 的值与指定可生产的媒体类型中有相同的时候，请求才会被匹配。而且，使用 *produces* 条件可以确保用于生成响应（response）的内容与指定的可生产的媒体类型是相同的。举个例子：

```
@Controller
@RequestMapping(path = "/pets/{petId}", method = RequestMethod.GET, produces = MediaTy
pe.APPLICATION_JSON_UTF8_VALUE)
@ResponseBody
public Pet getPet(@PathVariable String petId, Model model) {
    // 方法实现省略
}
```

> [Original] Be aware that the media type specified in the *produces* condition can also optionally specify a character set. For example, in the code snippet above we specify the same media type than the default one configured in `MappingJackson2HttpMessageConverter`, including the `UTF-8` charset.
>
> 要注意的是，通过 *condition* 条件指定的媒体类型也可以指定字符集。比如在上面的小段代码中，我们还是覆写了 `MappingJackson2HttpMessageConverter` 类中默认配置的媒体类型，同时，还指定了使用 `UTF-8` 的字符集。
>
> [Original] Just like with *consumes*, producible media type expressions can be negated as in *!text/plain* to match to all requests other than those with an *Accept* header value of *text/plain*. Also consider using constants provided in `MediaType` such as `APPLICATION_JSON_VALUE` and `APPLICATION_JSON_UTF8_VALUE`.

与 *consumes* 条件类似，可生产的媒体类型表达式也可以使用否定。比如，可以使用 *!text/plain* 来匹配所有请求头 *Accept* 中不含 *text/plain* 的请求。同时，在 `MediaType` 类中还定义了一些常量，比如 `APPLICATION_JSON_VALUE` 、 `APPLICATION_JSON_UTF8_VALUE` 等，推荐更多地使用它们。

> [Original] The *produces* condition is supported on the type and on the method level. Unlike most other conditions, when used at the type level, method-level producible types override rather than extend type-level producible types.
>
> *produces* 属性提供的是方法级的类型支持。与其他属性不同，当在类型级使用时，方法级的消费类型将覆盖类型级的配置，而非继承关系。

请求参数与请求头的值

> [Original] You can narrow request matching through request parameter conditions such as `"myParam"`, `"!myParam"`, or `"myParam=myValue"`. The first two test for request parameter presence/absence and the third for a specific parameter value. Here is an example with a request parameter value condition:

你可以筛选请求参数的条件来缩小请求匹配范围，比如 `"myParam"` 、 `"!myParam"` 及 `"myParam=myValue"` 等。前两个条件用于筛选存在/不存在某些请求参数的请求，第三个条件筛选具有特定参数值的请求。下面有个例子，展示了如何使用请求参数值的筛选条件：

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

    @RequestMapping(path = "/pets/{petId}", method = RequestMethod.GET, params="myPara
m=myValue")
    public void findPet(@PathVariable String ownerId, @PathVariable String petId, Mode
l model) {
        // 实际实现省略
    }

}
```

> [Original] The same can be done to test for request header presence/absence or to match based on a specific request header value:

同样，你可以用相同的条件来筛选请求头的出现与否，或者筛选出一个具有特定值的请求头：

```
@Controller
@RequestMapping("/owners/{ownerId}")
public class RelativePathUriTemplateController {

    @RequestMapping(path = "/pets", method = RequestMethod.GET, headers="myHeader=myVa
lue")
    public void findPet(@PathVariable String ownerId, @PathVariable String petId, Mode
l model) {
        // 方法体实现省略
    }

}
```

> [Original] Although you can match to *Content-Type* and *Accept* header values using media type wild cards (for example *"content-type=text/*"* will match to *"text/plain"* and *"text/html"*), it is recommended to use the *consumes* and *produces* conditions respectively instead. They are intended specifically for that purpose.
>
> 尽管，你可以使用媒体类型的通配符（比如 *"content-type=text/*"*）来匹配请求头 *Content-Type*和 *Accept*的值，但我们更推荐独立使用 *consumes*和 *produces*条件来筛选各自的请求。因为它们就是专门为区分这两种不同的场景而生的。

21.3.3 定义@RequestMapping注解的处理方法(handler method)

使用 `@RequestMapping` 注解的处理方法可以拥有非常灵活的方法签名，它支持的方法参数及返回值类型将在接下来的小节讲述。大多数参数都可以任意的次序出现，除了唯一的一个例外： `BindingResult` 参数。这在下节也会详细描述。

> Spring 3.1中新增了一些类，用以增强注解了 `@RequestMapping` 的处理方法，分别是 `RequestMappingHandlerMapping` 类和 `RequestMappingHandlerAdapter` 类。我们鼓励使用这组新的类，如果要使用Spring 3.1及以后版本的新特性，这组类甚至是必须使用的。这些增强类在MVC的命名空间配置和MVC的Java编程方式配置中都是默认开启的，如果不是使用这两种方法，那么就需要显式地配置。

支持的方法参数类型

下面列出所有支持的方法参数类型：

- 请求或响应对象（Servlet API）。可以是任何具体的请求或响应类型的对象，比如， `ServletRequest` 或 `HttpServletRequest` 对象等。
- `HttpSession` 类型的会话对象（Servlet API）。使用该类型的参数将要求这样一个session的存在，因此这样的参数永不为 `null` 。

> 存取session可能不是线程安全的，特别是在一个Servlet的运行环境中。如果应用可能有多个请求同时并发存取一个session场景，请考虑将RequestMappingHandlerAdapter类中的"synchronizeOnSession"标志设置为"true"。

- `org.springframework.web.context.request.WebRequest` 或 `org.springframework.web.context.request.NativeWebRequest` 。允许存取一般的请求参数和请求/会话范围的属性（attribute），同时无需绑定使用Servlet/Portlet的API
- 当前请求的地区信息 `java.util.Locale` ，由已配置的最相关的地区解析器解析得到。在MVC的环境下，就是应用中配置的 `LocaleResolver` 或 `LocaleContextResolver`
- 与当前请求绑定的时区信息 `java.util.TimeZone` （java 6以上的版本）/ `java.time.ZoneId` （java 8），由 `LocaleContextResolver` 解析得到
- 用于存取请求正文的 `java.io.InputStream` 或 `java.io.Reader` 。该对象与通过Servlet API拿到的输入流/Reader是一样的
- 用于生成响应正文的 `java.io.OutputStream` 或 `java.io.Writer` 。该对象与通过Servlet API拿到的输出流/Writer是一样的
- `org.springframework.http.HttpMethod` 。可以拿到HTTP请求方法
- 包装了当前被认证用户信息的 `java.security.Principal`

- 带 `@PathVariable` 注解的方法参数，其存放了URI模板变量中的值。详见"URI模板变量"一节
- 带 `@MatrixVariable` 注解的方法参数，其存放了URI路径段中的键值对。详见"矩阵变量"一节
- 带 `@RequestParam` 注解的方法参数，其存放了Servlet请求中所指定的参数。参数的值会被转换成方法参数所声明的类型。详见"使用@RequestParam注解绑定请求参数至方法参数"一节
- 带 `@RequestHeader` 注解的方法参数，其存放了Servlet请求中所指定的HTTP请求头的值。参数的值会被转换成方法参数所声明的类型。详见"使用@RequestHeader注解映射请求头属性"一节.
- 带 `@RequestBody` 注解的参数，提供了对HTTP请求体的存取。参数的值通过 `HttpMessageConverter` 被转换成方法参数所声明的类型。详见"使用@RequestBody注解映射请求体"一节"
- 带 `@RequestPart` 注解的参数，提供了对一个"multipart/form-data请求块（request part）内容的存取。更多的信息请参考21.10.5 "处理客户端文件上传的请求"一节和21.10 "Spring对多部分文件上传的支持"一节
- `HttpEntity<?>` 类型的参数，其提供了对HTTP请求头和请求内容的存取。请求流是通过 `HttpMessageConverter` 被转换成entity对象的。详见"HttpEntity"一节
- `java.util.Map` / `org.springframework.io.Model` / `org.springframework.ui.ModelMap` 类型的参数，用以增强默认暴露给视图层的模型(model)的功能
- `org.springframework.web.servlet.mvc.support.RedirectAttributes` 类型的参数，用以指定重定向下要使用到的属性集以及添加flash属性（暂存在服务端的属性，它们会在下次重定向请求的范围中有效）。详见"向重定向请求传递参数"一节
- 命令或表单对象，它们用于将请求参数直接绑定到bean字段（可能是通过setter方法）。你可以通过 `@InitBinder` 注解和/或 `HanderAdapter` 的配置来定制这个过程的类型转换。具体请参考 `RequestMappingHandlerAdapter` 类 `webBindingInitializer` 属性的文档。这样的命令对象，以及其上的验证结果，默认会被添加到模型model中，键名默认是该命令对象类的类名——比如， `some.package.OrderAddress` 类型的命令对象就使用属性名 `orderAddress` 类获取。 `ModelAttribute` 注解可以应用在方法参数上，用以指定该模型所用的属性名
- `org.springframework.validation.Errors` / `org.springframework.validation.BindingResult` 验证结果对象，用于存储前面的命令或表单对象的验证结果（紧接其前的第一个方法参数）。
- `org.springframework.web.bind.support.SessionStatus` 对象，用以标记当前的表单处理已结束。这将触发一些清理操作： `@SessionAttributes` 在类级别注解的属性将被移除
- `org.springframework.web.util.UriComponentsBuilder` 构造器对象，用于构造当前请求URL相关的信息，比如主机名、端口号、资源类型（scheme）、上下文路径、servlet映射中的相对部分（literal part）等

在参数列表中，`Errors` 或 `BindingResult` 参数必须紧跟在其所绑定的验证对象后面。这是因为，在参数列表中允许有多于一个的模型对象，Spring会为它们创建不同的 `BindingResult` 实例。因此，下面这样的代码是不能工作的：

BindingResult与@ModelAttribute错误的参数次序

```
@RequestMapping(method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("pet") Pet pet, Model model, BindingResult
 result) { ... }
```

上例中，因为在模型对象 `Pet` 和验证结果对象 `BindingResult` 中间还插了一个 `Model` 参数，这是不行的。要达到预期的效果，必须调整一下参数的次序：

```
@RequestMapping(method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result, Mode
l model) { ... }
```

> 对于一些带有 `required` 属性的注解（比如 `@RequestParam` 、 `@RequestHeader` 等），JDK 1.8的 `java.util.Optional` 可以作为被它们注解的方法参数。在这种情况下，使用 `java.util.Optional` 与 `required=false` 的作用是相同的。

支持的方法返回类型

以下是handler方法允许的所有返回类型：

- `ModelAndView` 对象，其中model隐含填充了命令对象，以及注解了 `@ModelAttribute` 字段的存取器被调用所返回的值。
- `Model` 对象，其中视图名称默认由 `RequestToViewNameTranslator` 决定，model隐含填充了命令对象以及注解了 `@ModelAttribute` 字段的存取器被调用所返回的值
- `Map` 对象，用于暴露model，其中视图名称默认由 `RequestToViewNameTranslator` 决定，model隐含填充了命令对象以及注解了 `@ModelAttribute` 字段的存取器被调用所返回的值
- `View` 对象。其中model隐含填充了命令对象，以及注解了 `@ModelAttribute` 字段的存取器被调用所返回的值。handler方法也可以增加一个 `Model` 类型的方法参数来增强model
- `String` 对象，其值会被解析成一个逻辑视图名。其中，model将默认填充了命令对象以及注解了 `@ModelAttribute` 字段的存取器被调用所返回的值。handler方法也可以增加一个 `Model` 类型的方法参数来增强model
- `void` 。如果处理器方法中已经对response响应数据进行了处理（比如在方法参数中定义一个 `ServletResponse` 或 `HttpServletResponse` 类型的参数并直接向其响应体中写东西），那么方法可以返回void。handler方法也可以增加一个 `Model` 类型的方法参数来增强model
- 如果处理器方法注解了 `ResponseBody` ，那么返回类型将被写到HTTP的响应体中，而返回

值会被 `HttpMessageConverters` 转换成所方法声明的参数类型。详见使用"@ResponseBody注解映射响应体"一节

- `HttpEntity<?>` 或 `ResponseEntity<?>` 对象，用于提供对Servlet HTTP响应头和响应内容的存取。对象体会被 `HttpMessageConverters` 转换成响应流。详见使用HttpEntity一节
- `HttpHeaders` 对象，返回一个不含响应体的response
- `Callable<?>` 对象。当应用希望异步地返回方法值时使用，这个过程由Spring MVC自身的线程来管理
- `DeferredResult<?>` 对象。当应用希望方法的返回值交由线程自身决定时使用
- `ListenableFuture<?>` 对象。当应用希望方法的返回值交由线程自身决定时使用
- `ResponseBodyEmitter` 对象，可用它异步地向响应体中同时写多个对象，also supported as the body within a `ResponseEntity`
- `SseEmitter` 对象，可用它异步地向响应体中写服务器端事件（Server-Sent Events）,also supported as the body within a `ResponseEntity`
- `StreamingResponseBody` 对象，可用它异步地向响应对象的输出流中写东西。also supported as the body within a `ResponseEntity`
- 其他任何返回类型，都会被处理成model的一个属性并返回给视图，该属性的名称为方法级的 `@ModelAttribute` 所注解的字段名（或者以返回类型的类名作为默认的属性名）。model隐含填充了命令对象以及注解了 `@ModelAttribute` 字段的存取器被调用所返回的值

使用@RequestParam将请求参数绑定至方法参数

你可以使用 `@RequestParam` 注解将请求参数绑定到你控制器的方法参数上。

下面这段代码展示了它的用法：

```
@Controller
@RequestMapping("/pets")
@SessionAttributes("pet")
public class EditPetForm {
    // ...
    @RequestMapping(method = RequestMapping.GET)
    public String setupForm(@RequestParam("petId") int petId, ModelMap model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }

    // ,..
}
```

若参数使用了该注解，则该参数默认是必须提供的，但你也可以把该参数标注为非必须的：只需要将 `@RequestParam` 注解的 `required` 属性设置为 `false` 即可（比如， `@RequestParam(path="id", required=false)` ）。

若所注解的方法参数类型不是 `String` ，则类型转换会自动地发生。详见"方法参数与类型转换"一节

若 `@RequestParam` 注解的参数类型是 `Map<String, String>` 或者 `MultiValueMap<String, String>` ，则该Map中会自动填充所有的请求参数。

使用@RequestBody注解映射请求体

方法参数中的 `@RequestBody` 注解暗示了方法参数应该被绑定了HTTP请求体的值。举个例子：

```
@RequestMapping(path = "/something", method = RequestMethod.PUT)
public void handle(@RequestBody String body, Writer writer) throws IOException {
    writer.write(body);
}
```

请求体到方法参数的转换是由 `HttpMessageConverter` 完成的。 `HttpMessageConverter` 负责将HTTP请求信息转换成对象，以及将对象转换回一个HTTP响应体。对于 `@RequestBody` 注解， `RequestMappingHandlerAdapter` 提供了以下几种默认的 `HttpMessageConverter` 支持：

- `ByteArrayHttpMessageConverter` 用以转换字节数组
- `StringHttpMessageConverter` 用以转换字符串
- `FormHttpMessageConverter` 用以将表格数据转换成 `MultiValueMap<String, String>` 或从 `MultiValueMap<String, String>` 中转换出表格数据
- `SourceHttpMessageConverter` 用于 `javax.xml.transform.Source` 类的互相转换

关于这些转换器的更多信息，请参考"HTTP信息转换器"一节。另外，如果使用的是MVC命名空间或Java编程的配置方式，会有更多默认注册的消息转换器。更多信息，请参考"启用MVC Java编程配置或MVC XML命令空间配置"一节。

若你更倾向于阅读和编写XML文件，那么你需要配置一个 `MarshallingHttpMessageConverter` 并为其提供 `org.springframework.oxm` 包下的一个 `Marshaller` 和 `Unmarshaller` 实现。下面的示例就为你展示如何直接在配置文件中配置它。但如果你的应用是使用MVC命令空间或MVC Java编程的方式进行配置的，则请参考"启用MVC Java编程配置或MVC XML命令空间配置"这一节。

```
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandl
erAdapter">
    <property name="messageConverters">
        <util:list id="beanList">
            <ref bean="stringHttpMessageConverter"/>
            <ref bean="marshallingHttpMessageConverter"/>
        </util:list>
    </property
</bean>

<bean id="stringHttpMessageConverter" class="org.springframework.http.converter.String
HttpMessageConverter"/>

<bean id="marshallingHttpMessageConverter"
        class="org.springframework.http.converter.xml.MarshallingHttpMessageConverter"
>
    <property name="marshaller" ref="castorMarshaller"/>
    <property name="unmarshaller" ref="castorMarshaller"/>
</bean>

<bean id="castorMarshaller" class="org.springframework.oxm.castor.CastorMarshaller"/>
```

注解了 `@RequestBody` 的方法参数还可以被 `@Valid` 注解，这样框架会使用已配置的 `Validator` 实例来对该参数进行验证。若你的应用是使用MVC命令空间或MVC Java编程的方式配置的，框架会假设在classpath路径下存在一个符合JSR-303规范的验证器，并自动将其作为默认配置。

与 `@ModelAttribute` 注解的参数一样， `Errors` 也可以被传入为方法参数，用于检查错误。如果没有声明这样一个参数，那么程序会抛出一个 `MethodArgumentNotValidException` 异常。该异常默认由 `DefaultHandlerExceptionResolver` 处理，处理程序会返回一个 `400` 错误给客户端。

> 关于如何通过MVC命令空间或MVC Java编程的方式配置消息转换器和验证器，也请参考"启用MVC Java编程配置或MVC XML命令空间配置"一节。

使用@ResponseBody注解映射响应体

`@ResponseBody` 注解与 `@RequestBody` 注解类似。 `@ResponseBody` 注解可被应用于方法上，标志该方法的返回值（更正，原文是return type，看起来应该是返回值）应该被直接写回到HTTP响应体中去（而不会被被放置到Model中或被解释为一个视图名）。举个例子：

```
@RequestMapping(path = "/something", method = RequestMethod.PUT)
@ResponseBody
public String helloWorld() {
    return "Hello World"
}
```

上面的代码结果是文本 `Hello World` 将被写入HTTP的响应流中。

与 `@RequestBody` 注解类似，Spring使用了一个 `HttpMessageConverter` 来将返回对象转换到响应体中。关于这些转换器的更多信息，请参考"HTTP信息转换器"一节。

使用@RestController注解创建REST控制器

当今让控制器实现一个REST API是非常常见的，这种场景下控制器只需要提供JSON、XML或其他自定义的媒体类型内容即可。你不需要在每个 `@RequestMapping` 方法上都增加一个 `@ResponseBody` 注解，更简明的做法是，给你的控制器加上一个 `@RestController` 的注解。

`@RestController` 是一个原生内置的注解，它结合了 `@ResponseBody` 与 `@Controller` 注解的功能。不仅如此，它也让你的控制器更表义，而且在框架未来的发布版本中，它也可能承载更多的意义。

与普通的 `@Controller` 无异， `@RestController` 也可以与 `@ControllerAdvice` bean配合使用。更多细节，请见使用@ControllerAdvice辅助控制器。

使用HTTP实体HttpEntity

`HttpEntity` 与 `@RequestBody` 和 `@ResponseBody` 很相似。除了能获得请求体和响应体中的内容之外， `HttpEntity` （以及专门负责处理响应的 `ResponseEntity` 子类）还可以存取请求头和响应头，像下面这样：

```
@RequestMapping("/something")
public ResponseEntity<String> handle(HttpEntity<byte[]> requestEntity) throws Unsuppor
tedEncodingException {
    String requestHeader = requestEntity.getHeaders().getFirst("MyRequestHeader");
    byte[] requestBody = requestEntity.getBody();

    // do something with request header and body

    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.set("MyResponseHeader", "MyValue");
    return new ResponseEntity<String>("Hello World", responseHeaders, HttpStatus.CREAT
ED);
}
```

上面这段示例代码先是获取了 `MyRequestHeader` 请求头的值，然后读取请求体的主体内容。读完以后往影响头中添加了一个自己的响应头 `MyResponseHeader` ，然后向响应流中写了字符串 `Hello World` ，最后把响应状态码设置为201（创建成功）。

与 `@RequestBody` 与 `@ResponseBody` 注解一样，Spring使用了 `HttpMessageConverter` 来对请求流和响应流进行转换。关于这些转换器的更多信息，请阅读上一小节以及"HTTP信息转换器"这一节。

对方法使用@ModelAttribute注解

`@ModelAttribute` 注解可被应用在方法或方法参数上。本节将介绍其被注解于方法上时的用法，下节会介绍其被用于注解方法参数的用法。

注解在方法上的 `@ModelAttribute` 说明了方法的作用是用于添加一个或多个属性到model上。这样的方法能接受与 `@RequestMapping` 注解相同的参数类型，只不过不能直接被映射到具体的请求上。在同一个控制器中，注解了 `@ModelAttribute` 的方法实际上会在 `@RequestMapping` 方法之前被调用。以下是几个例子：

```
// Add one attribute
// The return value of the method is added to the model under the name "account"
// You can customize the name via @ModelAttribute("myAccount")

@ModelAttribute
public Account addAccount(@RequestParam String number) {
    return accountManager.findAccount(number);
}

// Add multiple attributes

@ModelAttribute
public void populateModel(@RequestParam String number, Model model) {
    model.addAttribute(accountManager.findAccount(number));
    // add more ...
}
```

`@ModelAttribute` 方法通常被用来填充一些公共需要的属性或数据，比如一个下拉列表所预设的几种状态，或者宠物的几种类型，或者去取得一个HTML表单渲染所需要的命令对象，比如 `Account` 等。

留意 `@ModelAttribute` 方法的两种风格。在第一种写法中，方法通过返回值的方式默认地将添加一个属性；在第二种写法中，方法接收一个 `Model` 对象，然后可以向其中添加任意数量的属性。你可以在根据需要，在两种风格中选择合适的一种。

一个控制器可以拥有数量不限的 `@ModelAttribute` 方法。同个控制器内的所有这些方法，都会在 `@RequestMapping` 方法之前被调用。

`@ModelAttribute` 方法也可以定义在 `@ControllerAdvice` 注解的类中，并且这些 `@ModelAttribute` 可以同时对许多控制器生效。具体的信息可以参考使用@ControllerAdvice辅助控制器。

> 属性名没有被显式指定的时候又当如何呢？在这种情况下，框架将根据属性的类型给予一个默认名称。举个例子，若方法返回一个 `Account` 类型的对象，则默认的属性名为"account"。你可以通过设置 `@ModelAttribute` 注解的值来改变默认值。当向 `Model` 中直接添加属性时，请使用合适的重载方法 `addAttribute(..)` -即，带或不带属性名的方法。

`@ModelAttribute` 注解也可以被用在 `@RequestMapping` 方法上。这种情况下， `@RequestMapping` 方法的返回值将会被解释为model的一个属性，而非一个视图名。此时视图名将以视图命名约定来方式来决议，与返回值为void的方法所采用的处理方法类似——请见视图：请求与视图名的对应。

在方法参数上使用@ModelAttribute注解

如上一小节所解释， `@ModelAttribute` 注解既可以被用在方法上，也可以被用在方法参数上。这一小节将介绍它注解在方法参数上时的用法。

注解在方法参数上的 `@ModelAttribute` 说明了该方法参数的值将由model中取得。如果model中找不到，那么该参数会先被实例化，然后被添加到model中。在model中存在以后，请求中所有名称匹配的参数都会填充到该参数中。这在Spring MVC中被称为数据绑定，一个非常有用的特性，节约了你每次都需要手动从表格数据中转换这些字段数据的时间。

```
@RequestMapping(path = "/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.P
OST)
public String processSubmit(@ModelAttribute Pet pet) { }
```

以上面的代码为例，这个Pet类型的实例可能来自哪里呢？有几种可能:

- 它可能因为 `@SessionAttributes` 注解的使用已经存在于model中——详见"在请求之间使用@SessionAttributes注解，使用HTTP会话保存模型数据"一节
- 它可能因为在同个控制器中使用了 `@ModelAttribute` 方法已经存在于model中——正如上一小节所叙述的
- 它可能是由URI模板变量和类型转换中取得的（下面会详细讲解）
- 它可能是调用了自身的默认构造器被实例化出来的

`@ModelAttribute` 方法常用于从数据库中取一个属性值，该值可能通过 `@SessionAttributes` 注解在请求中间传递。在一些情况下，使用URI模板变量和类型转换的方式来取得一个属性是更方便的方式。这里有个例子：

```
@RequestMapping(path = "/accounts/{account}", method = RequestMethod.PUT)
public String save(@ModelAttribute("account") Account account) {

}
```

上面这个例子中，model属性的名称（"account"）与URI模板变量的名称相匹配。如果你配置了一个可以将 `String` 类型的账户值转换成 `Account` 类型实例的转换器 `Converter<String, Account>` ，那么上面这段代码就可以工作的很好，而不需要再额外写一个 `@ModelAttribute` 方法。

下一步就是数据的绑定。 `WebDataBinder` 类能将请求参数——包括字符串的查询参数和表单字段等——通过名称匹配到model的属性上。成功匹配的字段在需要的时候会进行一次类型转换（从String类型到目标字段的类型），然后被填充到model对应的属性中。数据绑定和数据验证的问题在第8章 验证，数据绑定和类型转换中提到。如何在控制器层来定制数据绑定的过程，在这一节 "定制WebDataBinder的初始化"中提及。

进行了数据绑定后，则可能会出现一些错误，比如没有提供必须的字段、类型转换过程的错误等。若想检查这些错误，可以在注解了 `@ModelAttribute` 的参数紧跟着声明一个 `BindingResult` 参数：

```
@RequestMapping(path = "/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.P
OST)
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) {
    if (result.hasErrors()) {
        return "petForm";
    }

    // ...

}
```

拿到 `BindingResult` 参数后，你可以检查是否有错误。有时你可以通过Spring的 `<errors>` 表单标签来在同一个表单上显示错误信息。

`BindingResult` 被用于记录数据绑定过程的错误，因此除了数据绑定外，你还可以把该对象传给自己定制的验证器来调用验证。这使得数据绑定过程和验证过程出现的错误可以被搜集到一处，然后一并返回给用户：

```
@RequestMapping(path = "/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.P
OST)
public String processSubmit(@ModelAttribute("pet") Pet pet, BindingResult result) {

    new PetValidator().validate(pet, result);
    if (result.hasErrors()) {
        return "petForm";
    }

    // ...

}
```

又或者，你可以通过添加一个JSR-303规范的 `@Valid` 注解，这样验证器会自动被调用。

```
@RequestMapping(path = "/owners/{ownerId}/pets/{petId}/edit", method = RequestMethod.POST)
public String processSubmit(@Valid @ModelAttribute("pet") Pet pet, BindingResult result) {

    if (result.hasErrors()) {
        return "petForm";
    }

    // ...

}
```

关于如何配置并使用验证，可以参考第8.8小节 "Spring验证"和第8章 验证，数据绑定和类型转换。

在请求之间使用@SessionAttributes注解，使用HTTP会话保存模型数据

类型级别的 `@SessionAttributes` 注解声明了某个特定处理器所使用的会话属性。通常它会列出该类型希望存储到session或converstaion中的model属性名或model的类型名，一般是用于在请求之间保存一些表单数据的bean。

以下的代码段演示了该注解的用法，它指定了模型属性的名称

```
@Controller
@RequestMapping("/editPet.do")
@SessionAttributes("pet")
public class EditPetForm {
    // ...
}
```

使用"application/x-www-form-urlencoded"数据

上一小节讲述了如何使用 `@ModelAttribute` 支持客户端浏览器的多次表单提交请求。对于不是使用的浏览器的客户端，我们也推荐使用这个注解来处理请求。但当请求是一个HTTP PUT方法的请求时，有一个事情需要注意。浏览器可以通过HTTP的GET方法或POST方法来提交表单数据，非浏览器的客户端还可以通过HTTP的PUT方法来提交表单。这就设计是个挑战，因为在Servlet规范中明确规定， `ServletRequest.getParameter*()` 系列的方法只能支持通过HTTP POST方法的方式提交表单，而不支持HTTP PUT的方式。

为了支持HTTP的PUT类型和PATCH类型的请求，Spring的 `spring-web` 模块提供了一个过滤器 `HttpPutFormContentFilter` 。你可以在 `web.xml` 文件中配置它：

```
    <filter>
        <filter-name>httpPutFormFilter</filter-name>
        <filter-class>org.springframework.web.filter.HttpPutFormContentFilter</filter-
class>
    </filter>

    <filter-mapping>
        <filter-name>httpPutFormFilter</filter-name>
        <servlet-name>dispatcherServlet</servlet-name>
    </filter-mapping>

    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class
>
    </servlet>
```

上面的过滤器将会拦截内容类型(content type)为 `application/x-www-form-urlencoded` 、HTTP方法为PUT或PATCH类型的请求，然后从请求体中读取表单数据，把它们包装在 `ServletRequest` 中。这是为了使表单数据能够通过 `ServletRequest.getParameter*()` 系列的方法来拿到。

> 因为 `HttpPutFormContentFilter` 会消费请求体的内容，因此，它不应该用于处理那些依赖于其他 `application/x-www-form-urlencoded` 转换器的PUT和PATCH请求，这包括了 `@RequestBodyMultiValueMap<String, String>` 和 `HttpEntity<MultiValueMap<String, String>>` 。

使用@CookieValue注解映射cookie值

`@CookieValue` 注解能将一个方法参数与一个HTTP cookie的值进行绑定。

看一个这样的场景：以下的这个cookie存储在一个HTTP请求中：

```
JSESSIONID=415A4AC178C59DACE0B2C9CA727CDD84
```

下面的代码演示了拿到 `JSESSIONID` 这个cookie值的方法：

```
@RequestMapping("/displayHeaderInfo.do")
public void displayHeaderInfo(@CookieValue("JSESSIONID") String cookie) {
    //...
}
```

若注解的目标方法参数不是 `String` 类型，则类型转换会自动进行。详见"方法参数与类型转换"一节。

这个注解可以注解到处理器方法上，在Servlet环境和Portlet环境都能使用。

使用 `@RequestHeader` 注解映射请求头属性

`@RequestHeader` 注解能将一个方法参数与一个请求头属性进行绑定。

以下是一个请求头的例子：

```
    Host                    localhost:8080
    Accept                  text/html,application/xhtml+xml,application/xml;q=0.9
    Accept-Language         fr,en-gb;q=0.7,en;q=0.3
    Accept-Encoding         gzip,deflate
    Accept-Charset          ISO-8859-1,utf-8;q=0.7,*;q=0.7
    Keep-Alive              300
```

以下的代码片段展示了如何取得 `Accept-Encoding` 请求头和 `Keep-Alive` 请求头的值：

```
@RequestMapping("/displayHeaderInfo.do")
public void displayHeaderInfo(@RequestHeader("Accept-Encoding") String encoding,
        @RequestHeader("Keep-Alive") long keepAlive) {
    //...
}
```

若注解的目标方法参数不是 `String` 类型，则类型转换会自动进行。"方法参数与类型转换"一节。

如果 `@RequestHeader` 注解应用在 `Map<String, String>` 、 `MultiValueMap<String, String>` 或 `HttpHeaders` 类型的参数上，那么所有的请求头属性值都会被填充到map中。

> Spring内置支持将一个逗号分隔的字符串（或其他类型转换系统所能识别的类型）转换成一个String类型的列表/集合。举个例子，一个注解了 `@RequestHeader("Accept")` 的方法参数可以是一个 `String` 类型，但也可以是 `String[]` 或 `List<String>` 类型的。

这个注解可以注解到处理器方法上，在Servlet环境和Portlet环境都能使用。

方法参数与类型转换

从请求参数、路径变量、请求头属性或者cookie中抽取出来的 `String` 类型的值，可能需要被转换成其所绑定的目标方法参数或字段的类型（比如，通过 `@ModelAttribute` 将请求参数绑定到方法参数上）。如果目标类型不是 `String` ，Spring会自动进行类型转换。所有的简单类型诸如int、long、Date都有内置的支持。如果想进一步定制这个转换过程，你可以通过 `WebDataBinder` （详见"定制WebDataBinder的初始化"一节），或者为 `Formatters` 配置一个 `FormattingConversionService` （详见8.6节 "Spring字段格式化"一节）来做到。

定制WebDataBinder的初始化

如果想通过Spring的 `WebDataBinder` 在属性编辑器中做请求参数的绑定，你可以使用在控制器内使用 `@InitBinder` 注解的方法、在注解了 `@ControllerAdvice` 的类中使用 `@InitBinder` 注解的方法，或者提供一个定制的 `WebBindingInitializer` 。更多的细节，请参考使用@ControllerAdvice辅助控制器一节。

数据绑定的定制：使用@InitBinder

使用 `@InitBinder` 注解控制器的方法，你可以直接在你的控制器类中定制应用的数据绑定。 `@InitBinder` 用来标记一些方法，这些方法会初始化一个 `WebDataBinder` 并用以为处理器方法填充命令对象和表单对象的参数。

除了命令/表单对象以及相应的验证结果对象，这样的“绑定器初始化”方法能够接收 `@RequestMapping` 所支持的所有参数类型。“绑定器初始化”方法不能有返回值，因此，一般将它们声明为 `void` 返回类型。特别地，当 `WebDataBinder` 与 `WebRequest` 或 `java.util.Locale` 一起作为方法参数时，你可以在代码中注册上下文相关的编辑器。

下面的代码示例演示了如何使用 `@InitBinder` 来配置一个 `CustomerDateEditor` ，后者会对所有 `java.util.Date` 类型的表单字段进行操作：

```
@Controller
public class MyFormController {

    @InitBinder
    public void initBinder(WebDataBinder binder) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(dateFormat, false
));
    }

    // ...
}
```

或者，你可以使用Spring 4.2提供的 `addCustomFormatter` 来指定 `Formatter` 的实现，而非通过 `PropertyEditor` 实例。这在你拥有一个需要 `Formatter` 的setup方法，并且该方法位于一个共享的 `FormattingConversionService` 中时非常有用。这样对于控制器级别的绑定规则的定制，代码更容易被复用。

```
@Controller
public class MyFormController {

    @InitBinder
    public void initBinder(WebDataBinder binder) {
        binder.addCustomFormatter(new DateFormatter("yyyy-MM-dd"));
    }

    // ...
}
```

配置定制的WebBindingInitializer

为了externalize数据绑定的初始化过程，你可以为 `WebBindingInitializer` 接口提供一个自己的实现，在其中你可以为 `AnnotationMethodHandlerAdapter` 提供一个默认的配置bean，以此来覆写默认的配置。

以下的代码来自PetClinic的应用，它展示了为 `WebBindingInitializer` 接口提供一个自定义实现： `org.springframework.samples.petclinic.web.ClinicBindingInitializer` 完整的配置过程。后者中配置了PetClinic应用中许多控制器所需要的属性编辑器PropertyEditors。

```
<bean class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandl
erAdapter">
    <property name="cacheSeconds" value="0"/>
    <property name="webBindingInitializer">
        <bean class="org.springframework.samples.petclinic.web.ClinicBindingInitialize
r"/>
    </property>
</bean>
```

`@InitBinder` 方法也可以定义在 `@ControllerAdvice` 注解的类上，这样配置可以为许多控制器所共享。这提供了除使用 `WebBindingInitializer` 外的另外一种方法。更多细节请参考使用@ControllerAdvice辅助控制器一节。

使用@ControllerAdvice辅助控制器

`@ControllerAdvice` 是一个组件注解，它使得其实现类能够被classpath扫描自动发现。若应用是通过MVC命令空间或MVC Java编程方式配置，那么该特性默认是自动开启的。

注解 `@ControllerAdvice` 的类可以拥有 `@ExceptionHandler` 、 `@InitBinder` 或 `@ModelAttribute` 注解的方法，并且这些方法会被应用至控制器类层次??的所有 `@RequestMapping` 方法上。

你也可以通过 `@ControllerAdvice` 的属性来指定其只对一个子集的控制器生效：

```
// Target all Controllers annotated with @RestController
@ControllerAdvice(annotations = RestController.class)
public class AnnotationAdvice {}

// Target all Controllers within specific packages
@ControllerAdvice("org.example.controllers")
public class BasePackageAdvice {}

// Target all Controllers assignable to specific classes
@ControllerAdvice(assignableTypes = {ControllerInterface.class, AbstractController.cla
ss})
public class AssignableTypesAdvice {}
```

更多的细节，请查阅 `@ControllerAdvice` 的文档。

下面两节，还看不太懂，待译。

Jackson Serialization View Support

It can sometimes be useful to filter contextually the object that will be serialized to the HTTP response body. In order to provide such capability, Spring MVC has built-in support for rendering with Jackson's Serialization Views.

To use it with an `@ResponseBody` controller method or controller methods that return `ResponseEntity`, simply add the `@JsonView` annotation with a class argument specifying the view class or interface to be used:

```
_@RestController_
public class UserController {

    _@RequestMapping(path = "/user", method = RequestMethod.GET)_
    _@JsonView(User.WithoutPasswordView.class)_
    public User getUser() {
        return new User("eric", "7!jd#h23");
    }
}

public class User {

    public interface WithoutPasswordView {};
    public interface WithPasswordView extends WithoutPasswordView {};

    private String username;
    private String password;

    public User() {
    }

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    _@JsonView(WithoutPasswordView.class)_
    public String getUsername() {
        return this.username;
    }

    _@JsonView(WithPasswordView.class)_
    public String getPassword() {
        return this.password;
    }
}
```

	Note

Note that despite `@JsonView` allowing for more than one class to be specified, the use on a controller method is only supported with exactly one class argument. Consider the use of a composite interface if you need to enable multiple views.

For controllers relying on view resolution, simply add the serialization view class to the model:

```
_@Controller_
public class UserController extends AbstractController {

    _@RequestMapping(path = "/user", method = RequestMethod.GET)_
    public String getUser(Model model) {
        model.addAttribute("user", new User("eric", "7!jd#h23"));
        model.addAttribute(JsonView.class.getName(), User.WithoutPasswordView.class);
        return "userView";
    }
}
```

Jackson JSONP Support

In order to enable JSONP support for `@ResponseBody` and `ResponseEntity` methods, declare an `@ControllerAdvice` bean that extends `AbstractJsonpResponseBodyAdvice` as shown below where the constructor argument indicates the JSONP query parameter name(s):

```
_@ControllerAdvice_
public class JsonpAdvice extends AbstractJsonpResponseBodyAdvice {

    public JsonpAdvice() {
        super("callback");
    }
}
```

For controllers relying on view resolution, JSONP is automatically enabled when the request has a query parameter named `jsonp` or `callback`. Those names can be customized through `jsonpParameterNames` property.

21.3.4 异步请求的处理

Spring MVC 3.2开始引入了基于Servlet 3的异步请求处理。相比以前，控制器方法已经不一定需要返回一个值，而是可以返回一个 `java.util.concurrent.Callable` 的对象，并通过Spring MVC所管理的线程来产生返回值。与此同时，Servlet容器的主线程则可以退出并释放其资源了，同时也允许容器去处理其他的请求。通过一个 `TaskExecutor` ，Spring MVC可以在另外的线程中调用 `Callable` 。当 `Callable` 返回时，请求再携带 `Callable` 返回的值，再次被分配到Servlet容器中恢复处理流程。以下代码给出了一个这样的控制器方法作为例子：

```
@RequestMapping(method=RequestMethod.POST)
public Callable<String> processUpload(final MultipartFile file) {

    return new Callable<String>() {
        public String call() throws Exception {
            // ...
            return "someView";
        }
    };

}
```

另一个选择，是让控制器方法返回一个 `DeferredResult` 的实例。这种场景下，返回值可以由任何一个线程产生，也包括那些不是由Spring MVC管理的线程。举个例子，返回值可能是为了响应某些外部事件所产生的，比如一条JMS的消息，一个计划任务，等等。以下代码给出了一个这样的控制器作为例子：

```
@RequestMapping("/quotes")
@ResponseBody
public DeferredResult<String> quotes() {
    DeferredResult<String> deferredResult = new DeferredResult<String>();
    // Save the deferredResult somewhere..
    return deferredResult;
}

// In some other thread...
deferredResult.setResult(data);
```

如果对Servlet 3.0的异步请求处理特性没有了解，理解这个特性可能会有点困难。因此，阅读一下前者的文档将会很有帮助。以下给出了这个机制运作背后的一些原理：

- 一个servlet请求 `ServletRequest` 可以通过调用 `request.startAsync()` 方法而进入异步模式。这样做的主要结果就是该servlet以及所有的过滤器都可以结束，但其响应（response）会留待异步处理结束后再返回

- 调用 `request.startAsync()` 方法会返回一个 `AsyncContext` 对象，可用它对异步处理进行进一步的控制和操作。比如说它也提供了一个与转向（forward）很相似的 `dispatch` 方法，只不过它允许应用恢复Servlet容器的请求处理进程
- `ServletRequest` 提供了获取当前 `DispatherType` 的方式，后者可以用来区别当前处理的是原始请求、异步分发请求、转向，或是其他类型的请求分发类型。

有了上面的知识，下面可以来看一下 `Callable` 的异步请求被处理时所依次发生的事件：

- 控制器先返回一个 `Callable` 对象
- Spring MVC开始进行异步处理，并把该 `Callable` 对象提交给另一个独立线程的执行器 `TaskExecutor` 处理
- `DispatcherServlet` 和所有过滤器都退出Servlet容器线程，但此时方法的响应对象仍未返回
- `Callable` 对象最终产生一个返回结果，此时Spring MVC会重新把请求分派回Servlet容器，恢复处理
- `DispatcherServlet` 再次被调用，恢复对 `Callable` 异步处理所返回结果的处理

对 `DeferredResult` 异步请求的处理顺序也非常类似，区别仅在于应用可以通过任何线程来计算返回一个结果：

- 控制器先返回一个 `DeferredResult` 对象，并把它存取在内存（队列或列表等）中以便存取
- Spring MVC开始进行异步处理
- `DispatcherServlet` 和所有过滤器都退出Servlet容器线程，但此时方法的响应对象仍未返回
- 由处理该请求的线程对 `DeferredResult` 进行设值，然后Spring MVC会重新把请求分派回Servlet容器，恢复处理
- `DispatcherServlet` 再次被调用，恢复对该异步返回结果的处理

关于引入异步请求处理的背景和原因，以及什么时候使用它、为什么使用异步请求处理等问题，你可以从这个系列的博客中了解更多信息。

异步请求的异常处理

若控制器返回的 `Callable` 在执行过程中抛出了异常，又会发生什么事情？简单来说，这与一般的控制器方法抛出异常是一样的。它会被正常的异常处理流程捕获处理。更具体地说呢，当 `Callable` 抛出异常时，Spring MVC会把一个 `Exception` 对象分派给Servlet容器进行处理，而不是正常返回方法的返回值，然后容器恢复对此异步请求异常的处理。若方法返回的是一个 `DeferredResult` 对象，你可以选择调 `Exception` 实例的 `setResult` 方法还是 `setErrorResult` 方法。

拦截异步请求

处理器拦截器 `HandlerInterceptor` 可以实现 `AsyncHandlerInterceptor` 接口拦截异步请求，因为在异步请求开始时，被调用的回调方法是该接口的 `afterConcurrentHandlingStarted` 方法，而非一般的 `postHandle` 和 `afterCompletion` 方法。

如果需要与异步请求处理的生命流程有更深入的集成，比如需要处理timeout的事件等，则 `HandlerInterceptor` 需要注册一个 `CallableProcessingInterceptor` 或 `DeferredResultProcessingInterceptor` 拦截器。具体的细节可以参考 `AsyncHandlerInterceptor` 类的Java文档。

`DeferredResult` 类还提供了 `onTimeout(Runnable)` 和 `onCompletion(Runnable)` 等方法，具体的细节可以参考 `DeferredResult` 类的Java文档。

`Callable` 需要请求过期(timeout)和完成后的拦截时，可以把它包装在一个 `WebAsyncTask` 实例中，后者提供了相关的支持。

HTTP streaming(不知道怎么翻)

如前所述，控制器可以使用 `DeferredResult` 或 `Callable` 对象来异步地计算其返回值，这可以用于实现一些有用的技术，比如 long polling技术，让服务器可以尽可能快地向客户端推送事件。

如果你想在一个HTTP响应中同时推送多个事件，怎么办？这样的技术已经存在，与"Long Polling"相关，叫"HTTP Streaming"。Spring MVC支持这项技术，你可以通过让方法返回一个 `ResponseBodyEmitter` 类型对象来实现，该对象可被用于发送多个对象。通常我们所使用的 `@ResponseBody` 只能返回一个对象，它是通过 `HttpMessageConverter` 写到响应体中的。

下面是一个实现该技术的例子：

```
@RequestMapping("/events")
public ResponseBodyEmitter handle() {
    ResponseBodyEmitter emitter = new ResponseBodyEmitter();
    // Save the emitter somewhere..
    return emitter;
}

// In some other thread
emitter.send("Hello once");

// and again later on
emitter.send("Hello again");

// and done at some point
emitter.complete();
```

`ResponseBodyEmitter` 也可以被放到 `ResponseEntity` 体里面使用，这可以对响应状态和响应头做一些定制。

Note that `ResponseBodyEmitter` can also be used as the body in a `ResponseEntity` in order to customize the status and headers of the response.

使用“服务器端事件推送”的HTTP Streaming

`SseEmitter` 是 `ResponseBodyEmitter` 的一个子类，提供了对服务器端事件推送的技术的支持。服务器端事件推送其实只是一种HTTP Streaming的类似实现，只不过它服务器端所推送的事件遵循了W3C Server-Sent Events规范中定义的事件格式。

“服务器端事件推送”技术正如其名，是用于由服务器端向客户端进行的事件推送。这在Spring MVC中很容易做到，只需要方法返回一个 `SseEmitter` 类型的对象即可。

需要注意的是，Internet Explorer并不支持这项服务器端事件推送的技术。另外，对于更大型的web应用及更精致的消息传输场景——比如在线游戏、在线协作、金融应用等——来说，使用Spring的WebSocket（包含SockJS风格的实时WebSocket）更成熟一些，因为它支持的浏览器范围非常广（包括IE），并且，对于一个以消息为中心的架构中，它为服务器端-客户端间的事件发布-订阅模型的交互提供了更高层级的消息模式（messaging patterns）的支持。

直接写回输出流OutputStream的HTTP Streaming

`ResponseBodyEmitter` 也允许通过 `HttpMessageConverter` 向响应体中支持写事件对象。这可能是最常见的情形，比如写返回的JSON数据的时候。但有时，跳过消息转换的阶段，直接把数据写回响应的输出流 `OutputStream` 可能更有效，比如文件下载这样的场景。这可以通过返回一个 `StreamingResponseBody` 类型的对象来实现。

以下是一个实现的例子：

```
@RequestMapping("/download")
public StreamingResponseBody handle() {
    return new StreamingResponseBody() {
        @Override
        public void writeTo(OutputStream outputStream) throws IOException {
            // write...
        }
    };
}
```

`ResponseBodyEmitter` 也可以被放到 `ResponseEntity` 体里面使用，这可以对响应状态和响应头做一些定制。

异步请求处理的相关配置

Servlet容器配置

对于那些使用 `web.xml` 配置文件的应用，请确保 `web.xml` 的版本更新到3.0：

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance http://java.sun.com/xml/
ns/javaee
                    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">

    ...

</web-app>
```

异步请求必须在 `web.xml` 将 `DispatcherServlet` 下的子元素 `<async-supported>true</async-supported>` 设置为true。此外，所有可能参与异步请求处理的过滤器 `Filter` 都必须配置为支持ASYNC类型的请求分派。在Spring框架中为过滤器启用支持ASYNC类型的请求分派应是安全的，因为这些过滤器一般都继承了基类 `OncePerRequestFilter` ，后者在运行时会检查该过滤器是否需要参与到异步分派的请求处理中。

以下是一个例子，展示了 `web.xml` 的配置：

```
    <web-app xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
                http://java.sun.com/xml/ns/javaee
                http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
        version="3.0">

        <filter>
            <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
            <filter-class>org.springframework.~.OpenEntityManagerInViewFilter</filter-
class>
            <async-supported>true</async-supported>
        </filter>

        <filter-mapping>
            <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
            <url-pattern>/*</url-pattern>
            <dispatcher>REQUEST</dispatcher>
            <dispatcher>ASYNC</dispatcher>
        </filter-mapping>

    </web-app>
```

如果应用使用的是Servlet 3规范基于Java编程的配置方式，比如通过`WebApplicationInitializer`，那么你也需要设置"asyncSupported"标志和ASYNC分派类型的支持，就像你在`web.xml`中所配置的一样。你可以考虑直接继承`AbstractDispatcherServletInitializer`或`AbstractAnnotationConfigDispatcherServletInitializer`来简化配置，它们都自动地为你设置了这些配置项，并使得注册`Filter`过滤器实例变得非常简单。

Spring MVC配置

MVC Java编程配置和MVC命名空间配置方式都提供了配置异步请求处理支持的选择。`WebMvcConfigurer`提供了`configureAsyncSupport`方法，而`<mvc:annotation-driven>`有一个子元素`<async-support>`，它们都用以为此提供支持。

这些配置允许你覆写异步请求默认的超时时间，在未显式设置时，它们的值与所依赖的Servlet容器是相关的（比如，Tomcat设置的超时时间是10秒）。你也可以配置用于执行控制器返回值`Callable`的执行器`AsyncTaskExecutor`。Spring强烈推荐你配置这个选项，因为Spring MVC默认使用的是普通的执行器`SimpleAsyncTaskExecutor`。MVC Java编程配置及MVC命名空间配置的方式都允许你注册自己的`CallableProcessingInterceptor`和`DeferredResultProcessingInterceptor`拦截器实例。

若你需要为特定的`DeferredResult`覆写默认的超时时间，你可以选用合适的构造方法来实现。类似，对于`Callable`返回，你可以把它包装在一个`WebAsyncTask`对象中，并使用合适的构造方法定义超时时间。`WebAsyncTask`类的构造方法同时也能接受一个任务执行器`AsyncTaskExecutor`类型的参数。

21.3.5 对控制器测试

`spring-test` 模块对测试控制器 `@Controller` 提供了最原生的支持。详见14.6 "Spring MVC测试框架"一节。

21.4 处理器映射（Handler Mappings）

在Spring的上个版本中，用户需要在web应用的上下文中定义一个或多个的 `HandlerMapping` bean，用以将进入容器的web请求映射到合适的处理器方法上。允许在控制器上添加注解后，通常你就不必这么做了，因为 `RequestMappingHandlerMapping` 类会自动查找所有注解了 `@RequestMapping` 的 `@Controller` 控制器bean。同时也请知道，所有继承自 `AbstractHandlerMapping` 的处理器方法映射 `HandlerMapping` 类都拥有下列的属性，你可以对它们进行定制：

- 一个 `interceptors` 列表，指示了应用其上的一个拦截器列表。处理器方法拦截器会在21.4.1小节 使用HandlerInterceptor拦截请求中讨论。
- `defaultHandler` ，生效的默认处理器，when this handler mapping does not result in a matching handler.
- `order` ，根据order（见 `org.springframework.core.Ordered` 接口）属性的值，Spring会对上下文可用的所有处理器映射进行排序，并应用第一个匹配成功的处理器
- `alwaysUseFullPath` （总是使用完整路径）。若设置为 `true` ，Spring将在当前Servlet上下文中总是使用完整路径来查找合适的处理器。若设置为 `false` （默认就为 `false` ），则使用当前Servlet的mapping路径。举个例子，若一个Servlet的mapping路径是 `/testing/*` ，并且 `alwaysUseFullPath` 属性被设置为 `true` ，此时用于查找处理器的路径将是 `/testing/viewPage.html` ；而若 `alwaysUseFullPath` 属性的值为 `false` ，则此时查找路径是 `/viewPage.html`
- `urlDecode` ，默认设置为 `true` （也是Spring 2.5的默认设置）。若你需要比较加密过的路径，则把此标志设为 `false` 。需要注意的是， `HttpServletRequest` 永远以未加密的方式存储Servlet路径。此时，该路径将无法匹配到加密过的路径

下面的代码展示了配置一个拦截器的方法：

```
<beans>
    <bean id="handlerMapping" class="org.springframework.web.servlet.mvc.method.annota
tion.RequestMappingHandlerMapping">
        <property name="interceptors">
            <bean class="example.MyInterceptor"/>
        </property>
    </bean>
<beans>
```

21.4.1 使用HandlerInterceptor拦截请求

Spring的处理器映射机制包含了处理器拦截器。拦截器在你需要为特定类型的请求应用一些功能时可能很有用，比如，检查用户身份等。

处理器映射处理过程配置的拦截器，必须实现 `org.springframework.web.servlet` 包下的 `HandlerInterceptor` 接口。这个接口定义了三个方法： `preHandle(..)` ，它在处理器实际执行 之前 会被执行； `postHandle(..)` ，它在处理器执行 完毕 以后被执行； `afterCompletion(..)` ，它在 整个请求处理完成 之后被执行。这三个方法为各种类型的前处理和后处理需求提供了足够的灵活性。

`preHandle(..)` 方法返回一个boolean值。你可以通过这个方法来决定是否继续执行处理链中的部件。当方法返回 `true` 时，处理器链会继续执行；若方法返回 `false` ， `DispatcherServlet` 即认为拦截器自身已经完成了对请求的处理（比如说，已经渲染了一个合适的视图），那么其余的拦截器以及执行链中的其他处理器就不会再被执行了。

拦截器可以通过 `interceptors` 属性来配置，该选项在所有继承了 `AbstractHandlerMapping` 的处理器映射类 `HandlerMapping` 都提供了配置的接口。如下面代码样例所示：

```
<beans>
    <bean id="handlerMapping" class="org.springframework.web.servlet.mvc.method.annota
tion.RequestMappingHandlerMapping">
        <property name="interceptors">
            <list>
                <ref bean="officeHoursInterceptor"/>
            </list>
        </property>
    </bean>

    <bean id="officeHoursInterceptor" class="samples.TimeBasedAccessInterceptor">
        <property name="openingTime" value="9"/>
        <property name="closingTime" value="18"/>
    </bean>
<beans>
```

```
package samples;

public class TimeBasedAccessInterceptor extends HandlerInterceptorAdapter {

    private int openingTime;
    private int closingTime;

    public void setOpeningTime(int openingTime) {
        this.openingTime = openingTime;
    }

    public void setClosingTime(int closingTime) {
        this.closingTime = closingTime;
    }

    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
            Object handler) throws Exception {
        Calendar cal = Calendar.getInstance();
        int hour = cal.get(HOUR_OF_DAY);
        if (openingTime <= hour && hour < closingTime) {
            return true;
        }
        response.sendRedirect("http://host.com/outsideOfficeHours.html");
        return false;
    }
}
```

在上面的例子中，所有被此处理器处理的请求都会被 `TimeBasedAccessInterceptor` 拦截器拦截。如果当前时间在工作时间以外，那么用户就会被重定向到一个HTML文件提示用户，比如显示“你只有在工作时间才可以访问本网站”之类的信息。

> 使用 `RequestMappingHandlerMapping` 时，实际的处理器是一个处理器方法 `HandlerMethod` 的实例，它标识了一个将被用于处理该请求的控制器方法。

如你所见，Spring的拦截器适配器 `HandlerInterceptorAdapter` 让继承 `HandlerInterceptor` 接口变得更简单了。

> 上面的例子中，所有控制器方法处理的请求都会被配置的拦截器先拦截到。如果你想进一步缩小拦截的URL范围，你可以通过MVC命名空间或MVC Java编程的方式来配置，或者，声明一个 `MappedInterceptor` 类型的bean实例来处理。具体请见 21.16.1 启用MVC Java编程配置或MVC命名空间配置一小节。

需要注意的是， `HandlerInterceptor` 的后拦截 `postHandle` 方法不一定总是适用于注解了 `@ResponseBody` 或 `ResponseEntity` 的方法。这些场景中， `HttpMessageConverter` 会在拦截器的 `postHandle` 方法被调之前就把信息写回响应中。这样拦截器就无法再改变响应了，比如要增加一个响应头之类的。如果有这种需求，请让你的应用实现 `ResponseBodyAdvice` 接口，并将其定义为一个 `@ControllerAdvice` bean或直接在 `RequestMappingHandlerMapping` 中配置。

21.5 视图解析

所有web应用的MVC框架都提供了视图相关的支持。Spring提供了一些视图解析器，它们让你能够在浏览器中渲染模型，并支持你自由选用适合的视图技术而不必与框架绑定到一起。Spring原生支持JSP视图技术、Velocity模板技术和XSLT视图等。你可以阅读文档的第22章视图技术一章，里面讨论了如何集成并使用许多独立的视图技术。

有两个接口在Spring处理视图相关事宜时至关重要，分别是视图解析器接口 `ViewResolver` 和视图接口本身 `View` 。视图解析器 `ViewResolver` 负责处理视图名与实际视图之间的映射关系。视图接口 `View` 负责准备请求，并将请求的渲染交给某种具体的视图技术实现。

21.5.1 使用ViewResolver接口解析视图

正如在21.3 控制器的实现一节中所讨论的，Spring MVC中所有控制器的处理器方法都必须返回一个逻辑视图的名字，无论是显式返回（比如返回一个 `String` 、 `View` 或者 `ModelAndView` ）还是隐式返回（比如基于约定的返回）。Spring中的视图由一个视图名标识，并由视图解析器来渲染。Spring有非常多内置的视图解析器。下表列出了大部分，表后也给出了一些例子。

表21.3 视图解析器

视图解析器	描述
`AbstractCachingViewResolver`	一个抽象的视图解析器类，提供了缓存视图的功能。通常视图在能够被使用之前需要经过准备。继承这个基类的视图解析器即可以获得缓存视图的能力。
`XmlViewResolver`	视图解析器接口 `ViewResolver` 的一个实现，该类接受一个XML格式的配置文件。该XML文件必须与Spring XML的bean工厂有相同的DTD。默认的配置文件名是 `/WEB-INF/views.xml` 。
`ResourceBundleViewResolver`	视图解析器接口 `ViewResolver` 的一个实现，采用bundle根路径所指定的 `ResourceBundle` 中的bean定义作为配置。一般bundle都定义在classpath路径下的一个配置文件中。默认的配置文件名为 `views.properties` 。
`UrlBasedViewResolver`	`ViewResolver` 接口的一个简单实现。它直接使用URL来解析到逻辑视图名，除此之外不需要其他任何显式的映射声明。如果你的逻辑视图名与你真正的视图资源名是直接对应的，那么这种直接解析的方式就很方便，不需要你再指定额外的映射。
`InternalResourceViewResolver`	`UrlBasedViewResolver` 的一个好用的子类。它支持内部资源视图（具体来说，Servlet和JSP）、以及诸如 `JstlView` 和 `TilesView` 等类的子类。You can specify the view class for all views generated by this resolver by using `setViewClass(..)` 。更多的细节，请见 `UrlBasedViewResolver` 类的java文档。
`VelocityViewResolver` / `FreeMarkerViewResolver`	`UrlBasedViewResolver` 下的实用子类，支持Velocity视图 `VelocityView` （Velocity模板）和FreeMarker视图 `FreeMarkerView` 以及它们对应子类。
`ContentNegotiatingViewResolver`	视图解析器接口 `ViewResolver` 的一个实现，它会根据所请求的文件名或请求的 `Accept` 头来解析一个视图。更多细节请见21.5.4 内容协商视图解析器"ContentNegotiatingViewResolver"一小节。

我们可以举个例子，假设这里使用的是JSP视图技术，那么我们可以使用一个基于URL的视图解析器 `UrlBasedViewResolver` 。这个视图解析器会将URL解析成一个视图名，并将请求转交给请求分发器来进行视图渲染。

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolv
er">
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

若返回一个 `test` 逻辑视图名，那么该视图解析器会将请求转发到 `RequestDispatcher` ，后者会将请求交给 `/WEB-INF/jsp/test.jsp` 视图去渲染。

如果需要在应用中使用多种不同的视图技术，你可以使用 `ResourceBundleViewResolver` ：

```
<bean id="viewResolver"
        class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="views"/>
    <property name="defaultParentView" value="parentView"/>
</bean>
```

`ResourceBundleViewResolver` 会检索由bundle根路径下所配置的 `ResourceBundle` ，对于每个视图而言，其视图类由 `[viewname].(class)` 属性的值指定，其视图url由 `[viewname].url` 属性的值指定。下一节将详细讲解视图技术，你可以在那里找到更多例子。你还可以看到，视图还允许有基视图，即properties文件中所有视图都“继承”的一个文件。通过继承技术，你可以为众多视图指定一个默认的视图基类。

> `AbstractCachingViewResolver` 的子类能够缓存已经解析过的视图实例。关闭缓存特性也是可以的，只需要将 `cache` 属性设置为 `false` 即可。另外，如果实在需要在运行时刷新某个视图（比如修改了Velocity模板时），你可以使用 `removeFromCache(String viewName, Locale loc)` 方法。`

21.5.2 视图链

Spring支持同时使用多个视图解析器。因此，你可以配置一个解析器链，并做更多的事比如，在特定条件下覆写一个视图等。你可以通过把多个视图解析器设置到应用上下文(application context)中的方式来串联它们。如果需要指定它们的次序，那么设置 `order` 属性即可。请记住，order属性的值越大，该视图解析器在链中的位置就越靠后。

在下面的代码例子中，视图解析器链中包含了两个解析器：一个是 `InternalResourceViewResolver` ，它总是自动被放置在解析器链的最后；另一个是 `XmlViewResolver` ，它用来指定Excel视图。 `InternalResourceViewResolver` 不支持Excel视图。

```
<bean id="jspViewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
</bean>

<bean id="excelViewResolver" class="org.springframework.web.servlet.view.XmlViewResolver">
    <property name="order" value="1"/>
    <property name="location" value="/WEB-INF/views.xml"/>
</bean>

<!-- in views.xml -->

<beans>
    <bean name="report" class="org.springframework.example.ReportExcelView"/>
</beans>
```

如果一个视图解析器不能返回一个视图，那么Spring会继续检查上下文中其他的视图解析器。此时如果存在其他的解析器，Spring会继续调用它们，直到产生一个视图返回为止。如果最后所有视图解析器都不能返回一个视图，Spring就抛出一个 `ServletException` 。

视图解析器的接口清楚声明了，一个视图解析器是可以返回null值的，这表示不能找到任何合适的视图。并非所有的视图解析器都这么做，但是也存在不得不如此的场景，即解析器确实无法检测对应的视图是否存在。比如， `InternalResourceViewResolver` 在内部使用了 `RequestDispatcher` ，并且进入分派过程是检测一个JSP视图是否存在的唯一方法，但这个过程仅可能发生唯一一次。同样的 `VelocityViewResolver` 和部分其他的视图解析器也存在这样的情况。具体的请查阅某个特定的视图解析器的Java文档，看它是否会report不存在的视图。因此，如果不把 `InternalResourceViewResolver` 放置在解析器链的最后，将可能导致解析器链无法完全执行，因为 `InternalResourceViewResolver` 永远都会 返回一个视图。

21.5.3 视图重定向

如前所述，控制器通常都会返回一个逻辑视图名，然后视图解析器会把它解析到一个具体的视图技术上去渲染。对于一些可以由Servlet或JSP引擎来处理的视图技术，比如JSP等，这个解析过程通常是由 `InternalResourceViewResolver` 和 `InternalResourceView` 协作来完成的，而这通常会调用Servlet的API `RequestDispatcher.forward(..)` 方法或 `RequestDispatcher.include(..)` 方法，并发生一次内部的转发（forward）或引用（include）。而对于其他的视图技术，比如Velocity、XSLT等，视图本身的内容是直接被写回响应流中的。

有时，我们想要在视图渲染之前，先把一个HTTP重定向请求发送回客户端。比如，当一个控制器成功地接受到了 `POST` 过来的数据，而响应仅仅是委托另一个控制器来处理（比如一次成功的表单提交）时，我们希望发生一次重定向。在这种场景下，如果只是简单地使用内部转发，那么意味着下一个控制器也能看到这次 `POST` 请求携带的数据，这可能导致一些潜在的问题，比如可能会与其他期望的数据混淆，等。此外，另一种在渲染视图前对请求进行重定向的需求是，防止用户多次提交表单的数据。此时若使用重定向，则浏览器会先发送第一个 `POST` 请求；请求被处理后浏览器会收到一个重定向响应，然后浏览器直接被重定向到一个不同的URL，最后浏览器会使用重定向响应中携带的URL发起一次 `GET` 请求。因此，从浏览器的角度看，当前所见的页面并不是 `POST` 请求的结果，而是一次 `GET` 请求的结果。这就防止了用户因刷新等原因意外地提交了多次同样的数据。此时刷新会重新 `GET` 一次结果页，而不是把同样的 `POST` 数据再发送一遍。

重定向视图 RedirectView

强制重定向的一种方法是，在控制器中创建并返回一个Spring重定向视图 `RedirectView` 的实例。它会使得 `DispatcherServlet` 放弃使用一般的视图解析机制，因为你已经返回一个（重定向）视图给 `DispatcherServlet` 了，所以它会构造一个视图来满足渲染的需求。紧接着 `RedirectView` 会调用 `HttpServletResponse.sendRedirect()` 方法，发送一个HTTP重定向响应给客户端浏览器。

如果你决定返回 `RedirectView` ，并且这个视图实例是由控制器内部创建出来的，那我们更推荐在外部配置重定向URL然后注入到控制器中来，而不是写在控制器里面。这样它就可以与视图名一起在配置文件中配置。关于如何实现这个解耦，请参考 重定向前缀——redirect:一小节。

向重定向目标传递数据

模型中的所有属性默认都会考虑作为URI模板变量被添加到重定向URL中。剩下的其他属性，如果是基本类型或者基本类型的集合或数组，那它们将被自动添加到URL的查询参数中去。如果model是专门为该重定向所准备的，那么把所有基本类型的属性添加到查询参数中可能是我们期望那个的结果。但是，在包含注解的控制器中，model可能包含了专门作为渲染用途的属性（比如一个下拉列表的字段值等）。为了避免把这样的属性也暴露在URL中，`@RequestMapping` 方法可以声明一个 `RedirectAttributes` 类型的方法参数，用它来指定专门供重定向视图 `RedirectView` 取用的属性。如果重定向成功发生，那么 `RedirectAttributes` 对象中的内容就会被使用；否则则使用模型model中的数据。

`RequestMappingHandlerAdapter` 提供了一个 `"ignoreDefaultModelOnRedirect"` 标志。它被用来标记默认 `Model` 中的属性永远不应该被用于控制器方法的重定向中。控制器方法应该声明一个 `RedirectAttributes` 类的参数。如果不声明，那就没有参数被传递到重定向的视图 `RedirectView` 中。在MVC命名空间或MVC Java编程配置方式中，为了维持向后的兼容性，这个标志都仍被保持为 `false`。但如果你的应用是一个新的项目，那么我们推荐把它的值设置成 `true`。

请注意，当前请求URI中的模板变量会在填充重定向URL的时候自动对应用可见，而不需要显式地在 `Model` 或 `RedirectAttributes` 中再添加属性。请看下面的例子：

```
@RequestMapping(path = "/files/{path}", method = RequestMethod.POST)
public String upload(...) {
    // ...
    return "redirect:files/{path}";
}
```

另外一种向重定向目标传递数据的方法是通过 闪存属性（*Flash Attributes*）。与其他重定向属性不同，flash属性是存储在HTTP session中的（因此不会出现在URL中）。更多内容，请参考21.6 使用闪存属性一节。

重定向前缀——redirect:

尽管使用 `RedirectView` 来做重定向能工作得很好，但如果控制器自身还是需要创建一个 `RedirectView`，那无疑控制器还是了解重定向这么一件事情的发生。这还是有点不尽完美，不同范畴的耦合还是太强。控制器其实不应该去关心响应会如何被渲染。In general it should operate only in terms of view names that have been injected into it.

一个特别的视图名前缀能完成这个解耦：`redirect:`。如果返回的视图名中含有 `redirect:` 前缀，那么 `UrlBasedViewResolver`（及它的所有子类）就会接受到这个信号，意识到这里需要发生重定向。然后视图名剩下的部分会被解析成重定向URL。

这种方式与通过控制器返回一个重定向视图`RedirectView`所达到的效果是一样的，不过这样一来控制器就可以只专注于处理并返回逻辑视图名了。如果逻辑视图名是这样的形式：`redirect:/myapp/some/resource`，他们重定向路径将以Servlet上下文作为相对路径进行查找，而逻辑视图名如果是这样的形式：`redirect:http://myhost.com/some/arbitrary/path`，那么重定向URL使用的就是绝对路径。

注意的是，如果控制器方法注解了`@ResponseStatus`，那么注解设置的状态码值会覆盖`RedirectView`设置的响应状态码值。

重定向前缀——forward:

对于最终会被`UrlBasedViewResolver`或其子类解析的视图名，你可以使用一个特殊的前缀：`forward:`。这会导致一个`InternalResourceView`视图对象的创建（它最终会调用`RequestDispatcher.forward()`方法），后者会认为视图名剩下的部分是一个URL。因此，这个前缀在使用`InternalResourceViewResolver`和`InternalResourceView`时并没有特别的作用（比如对于JSP来说）。但当你主要使用的是其他的视图技术，而又想要强制把一个资源转发给Servlet/JSP引擎进行处理时，这个前缀可能就很有用（或者，你也可能同时串联多个视图解析器）。

与`redirect:`前缀一样，如果控制器中的视图名使用了`forward:`前缀，控制器本身并不会发觉任何异常，它关注的仍然只是如何处理响应的问题。

21.5.4 内容协商解析器 ContentNegotiatingViewResolver

`ContentNegotiatingViewResolver` 自己并不会解析视图，而是委托给其他的视图解析器去处理。

The `ContentNegotiatingViewResolver` does not resolve views itself but rather delegates to other view resolvers, selecting the view that resembles the representation requested by the client. Two strategies exist for a client to request a representation from the server:

- Use a distinct URI for each resource, typically by using a different file extension in the URI. For example, the URI `<http://www.example.com/users/fred.pdf>` requests a PDF representation of the user fred, and `<http://www.example.com/users/fred.xml>` requests an XML representation.
- Use the same URI for the client to locate the resource, but set the `Accept` HTTP request header to list the media types that it understands. For example, an HTTP request for `<http://www.example.com/users/fred>` with an `Accept` header set to `application/pdf` requests a PDF representation of the user fred, while `<http://www.example.com/users/fred>` with an `Accept` header set to `text/xml` requests an XML representation. This strategy is known as content negotiation.

	Note

One issue with the `Accept` header is that it is impossible to set it in a web browser within HTML. For example, in Firefox, it is fixed to:

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

For this reason it is common to see the use of a distinct URI for each representation when developing browser based web applications.

To support multiple representations of a resource, Spring provides the `ContentNegotiatingViewResolver` to resolve a view based on the file extension or `Accept` header of the HTTP request. `ContentNegotiatingViewResolver` does not perform the view resolution itself but instead delegates to a list of view resolvers that you specify through the bean property `ViewResolvers` .

The `ContentNegotiatingViewResolver` selects an appropriate `View` to handle the request by comparing the request media type(s) with the media type (also known as `Content-Type`) supported by the `View` associated with each of its `ViewResolvers`. The first `View` in the list that has a compatible `Content- Type` returns the representation to the client. If a compatible view cannot be supplied by the `ViewResolver` chain, then the list of views specified through the `DefaultViews` property will be consulted. This latter option is appropriate for singleton `Views` that can render an appropriate representation of the current resource regardless of the logical view name. The `Accept` header may include wild cards, for example `text/*`, in which case a `View` whose Content-Type was `text/xml` is a compatible match.

To support custom resolution of a view based on a file extension, use a `ContentNegotiationManager`: see Section 21.16.6, "Content Negotiation".

Here is an example configuration of a `ContentNegotiatingViewResolver`:

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
    <property name="viewResolvers">
        <list>
            <bean class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
            <bean class="org.springframework.web.servlet.view.InternalResourceViewReso
lver">
                <property name="prefix" value="/WEB-INF/jsp/"/>
                <property name="suffix" value=".jsp"/>
            </bean>
        </list>
    </property>
    <property name="defaultViews">
        <list>
            <bean class="org.springframework.web.servlet.view.json.MappingJackson2Json
View"/>
        </list>
    </property>
</bean>

<bean id="content" class="com.foo.samples.rest.SampleContentAtomView"/>
```

The `InternalResourceViewResolver` handles the translation of view names and JSP pages, while the `BeanNameViewResolver` returns a view based on the name of a bean. (See " [Resolving views with the ViewResolver interface](mvc.html

mvc-viewresolver-resolver "21.5.1 Resolving views with the ViewResolver

interface")" for more details on how Spring looks up and instantiates a view.) In this example, the `content` bean is a class that inherits from `AbstractAtomFeedView`, which returns an Atom RSS feed. For more information on creating an Atom Feed representation, see the section Atom Views.

In the above configuration, if a request is made with an `.html` extension, the view resolver looks for a view that matches the `text/html` media type. The `InternalResourceViewResolver` provides the matching view for `text/html`. If the request is made with the file extension `.atom`, the view resolver looks for a view that matches the `application/atom+xml` media type. This view is provided by the `BeanNameViewResolver` that maps to the `SampleContentAtomView` if the view name returned is `content`. If the request is made with the file extension `.json`, the `MappingJackson2JsonView` instance from the `DefaultViews` list will be selected regardless of the view name. Alternatively, client requests can be made without a file extension but with the `Accept` header set to the preferred media-type, and the same resolution of request to views would occur.

	Note

If `ContentNegotiatingViewResolver's list of ViewResolvers is not configured explicitly, it automatically uses any ViewResolvers defined in the application context.

The corresponding controller code that returns an Atom RSS feed for a URI of the form `<http://localhost/content.atom>` or `<http://localhost/content>` with an `Accept` header of application/atom+xml is shown below.

```
@Controller
public class ContentController {

    private List<SampleContent> contentList = new ArrayList<SampleContent>();

    @RequestMapping(path="/content", method=RequestMethod.GET)
    public ModelAndView getContent() {
        ModelAndView mav = new ModelAndView();
        mav.setViewName("content");
        mav.addObject("sampleContentList", contentList);
        return mav;
    }

}
```

21.6 使用闪存属性FlashAttributes

Flash属性（flash attributes）提供了一个请求为另一个请求存储有用属性的方法。这在重定向的时候最常使用，比如常见的 *POST/REDIRECT/GET* 模式。Flash属性会在重定向前被暂时地保存起来（通常是保存在session中），重定向后会重新被下一个请求取用并立即从原保存地移除。

为支持flash属性，Spring MVC提供了两个抽象。`FlashMap` 被用来存储flash属性，而用 `FlashMapManager` 来存储、取回、管理 `FlashMap` 的实例。

对flash属性的支持默认是启用的，并不需要显式声明，不过没用到它时它绝不会主动地去创建HTTP会话（session）。对于每个请求，框架都会“传进”一个 `FlashMap` ，里面存储了从上个请求（如果有）保存下来的属性；同时，每个请求也会“输出”一个 `FlashMap` ，里面保存了要给下个请求使用的属性。两个 `FlashMap` 实例在Spring MVC应用中的任何地点都可以通过 `RequestContextUtils` 工具类的静态方法取得。

控制器通常不需要直接接触 `FlashMap` 。一般是通过 `@RequestMapping` 方法去接受一个 `RedirectAttributes` 类型的参数，然后直接地往其中添加flash属性。通过 `RedirectAttributes` 对象添加进去的flash属性会自动被填充到请求的“输出” `FlashMap` 对象中去。类似地，重定向后“传进”的 `FlashMap` 属性也会自动被添加到服务重定向URL的控制器参数 `Model` 中去。

匹配请求所使用的**flash**属性

flash属性的概念在其他许多的Web框架中也存在，并且实践证明有时可能会导致并发上的问题。这是因为从定义上讲，flash属性保存的时间是到下个请求接收到之前。问题在于，“下一个”请求不一定刚好就是你要重定向到的那个请求，它有可能是其他的异步请求（比如polling请求或者资源请求等）。这会导致flash属性在到达真正的目标请求前就被移除了。

为了减少这个问题发生的可能性，重定向视图 `RedirectView` 会自动为一个 `FlashMap` 实例记录其目标重定向URL的路径和查询参数。然后，默认的 `FlashMapManager` 会在为请求查找其该“传进”的 `FlashMap` 时，匹配这些信息。

这并不能完全解决重定向的并发问题，但极大程度地减少了这种可能性，因为它可以从重定向URL已有的信息中来做匹配。因此，一般只有在重定向的场景下，我们才推荐使用flash属性。

21.7 URI构造

在Spring MVC中，使用了 `UriComponentsBuilder` 和 `UriComponents` 两个类来提供一种构造和加密URI的机制。

比如，你可以通过一个URI模板字符串来填充并加密一个URI：

```
UriComponents uriComponents = UriComponentsBuilder.fromUriString(
        "http://example.com/hotels/{hotel}/bookings/{booking}").build();

URI uri = uriComponents.expand("42", "21").encode().toUri();
```

请注意 `UriComponents` 是不可变对象。因此 `expand()` 与 `encode()` 操作在必要的时候会返回一个新的实例。

你也可以使用一个URI组件实例对象来实现URI的填充与加密：

```
UriComponents uriComponents = UriComponentsBuilder.newInstance()
        .scheme("http").host("example.com").path("/hotels/{hotel}/bookings/{booking}")
.build()
        .expand("42", "21")
        .encode();
```

在Servlet环境下， `ServletUriComponentsBuilder` 类提供了一个静态的工厂方法，可以用于从Servlet请求中获取URL信息：

```
HttpServletRequest request = ...

// 主机名、schema, 端口号、请求路径和查询字符串都重用请求里已有的值
// 替换了其中的"accountId"查询参数

ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromRequest(request)
        .replaceQueryParam("accountId", "{id}").build()
        .expand("123")
        .encode();
```

或者，你也可以选择只复用请求中一部分的信息：

```
    // 重用主机名、端口号和context path
    // 在路径后添加"/accounts"

    ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromContextPath(requ
est)
            .path("/accounts").build()
```

或者，如果你的 `DispatcherServlet` 是通过名字（比如， `/main/*` ）映射请求的，you can also have the literal part of the servlet mapping included:

```
    // Re-use host, port, context path
    // Append the literal part of the servlet mapping to the path
    // Append "/accounts" to the path

    ServletUriComponentsBuilder ucb = ServletUriComponentsBuilder.fromServletMapping(r
equest)
            .path("/accounts").build()
```

21.7.1 为控制器和方法指定URI

Spring MVC也提供了构造指定控制器方法链接的机制。以下面代码为例子，假设我们有这样一个控制器：

```
@Controller
@RequestMapping("/hotels/{hotel}")
public class BookingController {

    @RequestMapping("/bookings/{booking}")
    public String getBooking(@PathVariable Long booking) {

    // ...

    }
}
```

你可以通过引用方法名字的办法来准备一个链接：

```
UriComponents uriComponents = MvcUriComponentsBuilder
    .fromMethodName(BookingController.class, "getBooking", 21).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();
```

在上面的例子中，我们为方法参数准备了填充值：一个long型的变量值21，以用于填充路径变量并插入到URL中。另外，我们还提供了一个值42，以用于填充其他剩余的URI变量，比如从类层级的请求映射中继承来的 `hotel` 变量。如果方法还有更多的参数，你可以为那些不需要参与URL构造的变量赋予null值。一般而言，只有 `@PathVariable` 和 `@RequestParam` 注解的参数才与URL的构造相关。

还有其他使用 `MvcUriComponentsBuilder` 的方法。比如，你可以通过类似mock掉测试对象的方法，用代理来避免直接通过名字引用一个控制器方法（以下方法假设 `MvcUriComponentsBuilder.on` 方法已经被静态导入）：

```
UriComponents uriComponents = MvcUriComponentsBuilder
    .fromMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();
```

上面的代码例子中使用了 `MvcUriComponentsBuilder` 类的静态方法。内部实现中，它依赖于 `ServletUriComponentsBuilder` 来从当前请求中抽取schema、主机名、端口号、context路径和servlet路径，并准备一个基本URL。大多数情况下它能良好工作，但有时还不行。比如，

在准备链接时，你可能在当前请求的上下文（context）之外（比如，执行一个准备链接links的批处理），或你可能需要为路径插入一个前缀（比如一个地区性前缀，它从请求中被移除，然后又重新被插入到链接中去）。

对于上面所提的场景，你可以使用重载过的静态方法 `fromXxx` ，它接收一个 `UriComponentsBuilder` 参数，然后从中获取基本URL以便使用。或你也可以使用一个基本URL创建一个 `MvcUriComponentsBuilder` 对象，然后使用实例对象的 `fromXxx` 方法。如下面的示例：

```
UriComponentsBuilder base = ServletUriComponentsBuilder.fromCurrentContextPath().path(
"/en");
MvcUriComponentsBuilder builder = MvcUriComponentsBuilder.relativeTo(base);
builder.withMethodCall(on(BookingController.class).getBooking(21)).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();
```

在视图中为控制器和方法指定URI

21.8 地区信息（Locales）

Spring的架构中的很多层面都提供了对国际化的支持，同样支持Spring MVC框架也能提供。`DispatcherServlet` 为你提供了自动使用用户的地区信息来解析消息的能力。而这，是通过 `LocaleResolver` 对象来完成的。

一个请求进入处理时，`DispatcherServlet` 会查找一个地区解析器。如果找到，就尝试使用它来设置地区相关的信息。通过调用 `RequestContext.getLocale()` 都能取到地区解析器所解析到的地区信息。

此外，如果你需要自动解析地区信息，你可以在处理器映射前加一个拦截器（关于更多处理器映射拦截器的知识，请参见21.4.1 使用HandlerInterceptor拦截请求一小节），并用它来根据条件或环境不同，比如，根据请求中某个参数值，来更改地区信息。

21.8.1 获取时区信息

除了获取客户端的地区信息外，有时他们所在的时区信息也非常有用。`LocaleContextResolver`接口为`LocaleResolver`提供了拓展点，允许解析器在`LocaleContext`中提供更多的信息，这里面就可以包含时区信息。

如果用户的时区信息能被解析到，那么你总可以通过`RequestContext.getTimeZone()`方法获得。时区信息会自动被Spring `ConversionService`下注册的日期/时间转换器`Converter`及格式化对象`Formatter`所使用。

21.8.2 Accept请求头解析器AcceptHeaderLocaleResolver

`AcceptHeaderLocaleResolver`解析器会检查客户端（比如，浏览器，等）所发送的请求中是否携带`accept-language`请求头。通常，该请求头字段中包含了客户端操作系统的地区信息。不过请注意，该解析器不支持时区信息的解析。

21.8.3 Cookie解析器CookieLocaleResolver

`CookieLocaleResolver` 解析会检查客户端是否有 `Cookie` ，里面可能存放了地区 `Locale` 或时区 `TimeZone` 信息。如果检查到相应的值，解析器就使用它们。通过该解析器的属性，你可以指定cookie的名称和其最大的存活时间。请见下面的例子，它展示了如何定义一个 `CookieLocaleResolver` ：

```
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleReso
lver">

    <property name="cookieName" value="clientlanguage"/>

    <!-- 单位为秒。若设置为-1，则cookie不会被持久化（客户端关闭浏览器后即被删除） -->
    <property name="cookieMaxAge" value="100000">

</bean>
```

表**21.4. CookieLocaleResolver**支持的属性

属性	默认值	描述
cookieName	classname + LOCALE	cookie名
cookieMaxAge	Integer.MAX_INT	cookie被保存在客户端的最长时间。如果该值为-1，那么cookie将不会被持久化，在客户端浏览器关闭之后就失效了
cookiePath	/	限制了cookie仅对站点下的某些特定路径可见。如果指定了cookiePath，那么cookie将仅对该路径及其子路径下的所有站点可见

21.8.4 Session解析器SessionLocaleResolver

`SessionLocaleResolver` 允许你从session中取得可能与用户请求相关联的地区 `Locale` 和时区 `TimeZone` 信息。与 `CookieLocaleResolver` 不同，这种存取策略仅将Servlet容器的 `HttpSession` 中相关的地区信息存取到本地。因此，这些设置仅会为该会话（session）临时保存，session结束后，这些设置就会失效。

不过请注意，该解析器与其他外部session管理机制，比如Spring的Session项目等，并没有直接联系。该 `SessionLocaleResolver` 仅会简单地从与当前请求 `HttpServletRequest` 相关的 `HttpSession` 对象中，取出对应的属性，并修改其值，仅此而已。

21.8.5 地区更改拦截器 LocaleChangeInterceptor

You can enable changing of locales by adding the `LocaleChangeInterceptor` to one of the handler mappings (see [Section 21.4, "Handler mappings"](mvc.html

mvc-handlermapping "21.4 Handler mappings")). It will detect a parameter in

the request and change the locale. It calls `setLocale()` on the `LocaleResolver` that also exists in the context. The following example shows that calls to all `*.view` resources containing a parameter named `siteLanguage` will now change the locale. So, for example, a request for the following URL, `<http://www.sf.net/home.view?siteLanguage=nl>` will change the site language to Dutch.

你可以在处理器映射（详见21.4 处理器映射（Handler mappings）小节）前添加一个 `LocaleChangeInterceptor` 拦截器来更改地区信息。它能检测请求中的参数，并根据其值相应地更新地区信息。它通过调用 `LocaleResolver` 的 `setLocale()` 方法来更改地区。下面的代码配置展示了如何为所有请求 `*.view` 路径并且携带了 `siteLanguage` 参数的资源请求更改地区。举个例子，一个URL为 `<http://www.sf.net/home.view?siteLanguage=nl>` 的请求将会将站点语言更改为荷兰语。

```
<bean id="localeChangeInterceptor" class="org.springframework.web.servlet.i18n.LocaleC
hangeInterceptor">
    <property name="paramName" value="siteLanguage"/>
</bean>

<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleReso
lver"/>

<bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerM
apping">
    <property name="interceptors">
        <list>
            <ref bean="localeChangeInterceptor"/>
        </list>
    </property>
    <property name="mappings">
        <value>/**/*.view=someController</value>
    </property>
</bean>
```

21.9 主题 themes

21.9.1 关于主题：概览

You can apply Spring Web MVC framework themes to set the overall look-and-feel of your application, thereby enhancing user experience. A theme is a collection of static resources, typically style sheets and images, that affect the visual style of the application.

你可以使用Spring Web MVC框架提供的主题来为整站的应用设置皮肤/主题（look-and-feel），这可以提高用户体验。主题 是指一系列静态资源的集合，并且主要是样式表和图片，它们决定了你的应用的视觉风格。

21.9.2 定义主题

要在你的应用中使用主题，你必须实现一个 `org.springframework.ui.context.ThemeSource` 接口。 `WebApplicationContext` 接口继承了 `ThemeSource` 接口，但主要的工作它还是委托给接口具体的实现来完成。默认的实现是 `org.springframework.ui.context.support.ResourceBundleThemeSource` ，它会从classpath的根路径下去加载配置文件。如果需要定制 `ThemeSource` 的实现，或要配置 `ResourceBundleThemeSource` 的基本前缀名（base name prefix），你可以在应用上下文（application context）下注册一个名字为保留名 `themeSource` 的bean，web应用的上下文会自动检测名字为 `themeSource` 的bean并使用它。

使用的是 `ResourceBundleThemeSource` 时，一个主题可以定义在一个简单的配置文件中。该配置文件会列出所有组成了该主题的资源。下面是个例子：

```
styleSheet=/themes/cool/style.css
background=/themes/cool/img/coolBg.jpg
```

属性的键（key）是主题元素在视图代码中被引用的名字。对于JSP视图来说，一般通过 `spring:theme` 这个定制化的标签（tag）来做，它与 `spring:message` 标签很相似。以下的JSP代码即使用了上段代码片段中定义的主题，用以定制整体的皮肤：

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
    <head>
        <link rel="stylesheet" href="<spring:theme code=''styleSheet''/>" type="text/c
ss"/>
    </head>
    <body style="background=<spring:theme code=''background''/>">
        ...
    </body>
</html>
```

默认情况下 `ResourceBundleThemeSource` 使用的基本名前缀（base name prefix）是空值。也即是说，配置文件是从根classpath路径下加载的。因此，你需要把主题的定义文件 `cool.properties` 放在classpath的根路径目录下，比如， `/WEB-INF/classes` 。 `ResourceBundleThemeSource` 采用了Java的标准资源bundle加载机制，完全支持国际化主题。比如，你可以创建一个 `/WEB-INF/classes/cool_nl.properties` 配置文件，并在其中引用一副有荷兰文的背景图片。

21.9.3 主题解析器

上一小节，我们讲了如何定义主题，定义之后，你要决定使用哪个主题。`DispatcherServlet`会查找一个名称为`themeResolver`的bean以确定使用哪个`ThemeResolver`的实现。主题解析器的工作原理与地区解析器`LocaleResolver`的工作原理大同小异。它会检测，对于一个请求来说，应该使用哪个主题，同时它也可以修改一个请求所应应用的主题。Spring提供了下列的这些主题解析器：

表**21.5. ThemeResolver**接口的实现

类名	描述
`FixedThemeResolver`	选择一个固定的主题，这是通过设置`defaultThemeName`这个属性值实现的
`SessionThemeResolver`	请求相关的主题保存在用户的HTTP会话（session）中。对于每个会话来说，它只需要被设置一次，但它不能在会话之间保存
`CookieThemeResolver`	选中的主题被保存在客户端的cookie中

Spring也提供了一个主题更改拦截器`ThemeChangeInterceptor`，以支持主题的更换。这很容易做到，只需要在请求中携带一个简单的请求参数即可。

21.10 Spring的multipart（文件上传）支持

21.10.1 概述

Spring内置对多路上传的支持，专门用于处理web应用中的文件上传。你可以通过注册一个可插拔的 `MultipartResolver` 对象来启用对文件多路上传的支持。该接口在定义于 `org.springframework.web.multipart` 包下。Spring为一般的文件上传提供了 `MultipartResolver` 接口的一个实现，为Servlet 3.0多路请求的转换提供了另一个实现。

默认情况下，Spring的多路上传支持是不开启的，因为有些开发者希望由自己来处理多路请求。如果想启用Spring的多路上传支持，你需要在web应用的上下文中添加一个多路传输解析器。每个进来的请求，解析器都会检查是不是一个多部分请求。若发现请求是完整的，则请求按正常流程被处理；如果发现请求是一个多路请求，则你在上下文中注册的 `MultipartResolver` 解析器会被用来处理该请求。之后，请求中的多路上传属性就与其他属性一样被正常对待了。【最后一句翻的不好，multipart翻译成多路还是多部分还在斟酌中。望阅读者注意此处。】

21.10.2 使用MultipartResolver与Commons FileUpload传输文件

下面的代码展示了如何使用一个通用的多路上传解析器 `CommonsMultipartResolver` ：

```
<bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsM
ultipartResolver">

    <!-- 支持的其中一个属性，支持的最大文件大小，以字节为单位 -->
    <property name="maxUploadSize" value="100000"/>

</bean>
```

当然，要让多路解析器正常工作，你需要在classpath路径下准备必须的jar包。如果使用的是通用的多路上传解析器 `CommonsMultipartResolver` ，你所需要的jar包是 `commons-fileupload.jar` 。

当Spring的 `DispatcherServlet` 检测到一个多部分请求时，它会激活你在上下文中声明的多路解析器并把请求交给它。解析器会把当前的 `HttpServletRequest` 请求对象包装成一个支持多路文件上传的请求对象 `MultipartHttpServletRequest` 。有了 `MultipartHttpServletRequest` 对象，你不仅可以获取该多路请求中的信息，还可以在你的控制器中获得该多路请求的内容本身。

21.10.3 Servlet 3.0下的MultipartResolver

要使用基于Servlet 3.0的多路传输转换功能，你必须在 `web.xml` 中为 `DispatcherServlet` 添加一个 `multipart-config` 元素，或者通过Servlet编程的方法使用 `javax.servlet.MultipartConfigElement` 进行注册，或你自己定制了自己的Servlet类，那你必须使用 `javax.servlet.annotation.MultipartConfig` 对其进行注解。其他诸如最大文件大小或存储位置等配置选项都必须在这个Servlet级别进行注册，因为Servlet 3.0不允许在解析器MultipartResolver的层级配置这些信息。

当你通过以上任一种方式启用了Servlet 3.0多路传输转换功能，你就可以把一个 `StandardServletMultipartResolver` 解析器添加到你的Spring配置中去了：

```
<bean id="multipartResolver" class="org.springframework.web.multipart.support.Standard
ServletMultipartResolver">
</bean>
```

21.10.4 处理表单中的文件上传

当解析器 `MultipartResolver` 完成处理时，请求便会像其他请求一样被正常流程处理。首先，创建一个接受文件上传的表单将允许用于直接上传整个表单。编码属性（ `enctype="multipart/form-data"` ）能让浏览器知道如何对多路上传请求的表单进行编码（encode）。

```
<html>
    <head>
        <title>Upload a file please</title>
    </head>
    <body>
        <h1>Please upload a file</h1>
        <form method="post" action="/form" enctype="multipart/form-data">
            <input type="text" name="name"/>
            <input type="file" name="file"/>
            <input type="submit"/>
        </form>
    </body>
</html>
```

下一步是创建一个能处理文件上传的控制器。这里需要的控制器与一般注解了 `@Controller` 的控制器基本一样，除了它接受的方法参数类型是 `MultipartHttpServletRequest` ，或 `MultipartFile` 。

```
@Controller
public class FileUploadController {

    @RequestMapping(path = "/form", method = RequestMethod.POST)
    public String handleFormUpload(@RequestParam("name") String name, @RequestParam("f
ile") MultipartFile file) {

        if (!file.isEmpty()) {
            byte[] bytes = file.getBytes();
            // store the bytes somewhere
            return "redirect:uploadSuccess";
        }

        return "redirect:uploadFailure";
    }

}
```

请留意 `@RequestParam` 注解是如何将方法参数对应到表单中的定义的输入字段的。在上面的例子中，我们拿到了 `byte[]` 文件数据，只是没对它做任何事。在实际应用中，你可能会将它保存到数据库、存储在文件系统上，或做其他的处理。

当使用Servlet 3.0的多路传输转换时，你也可以使用 `javax.servlet.http.Part` 作为方法参数：

```
@Controller
public class FileUploadController {

    @RequestMapping(path = "/form", method = RequestMethod.POST)
    public String handleFormUpload(@RequestParam("name") String name, @RequestParam("f
ile") Part file) {

        InputStream inputStream = file.getInputStream();
        // store bytes from uploaded file somewhere

        return "redirect:uploadSuccess";
    }

}
```

21.10.5 处理客户端发起的文件上传请求

在使用了RESTful服务的场景下，非浏览器的客户端也可以直接提交多路文件请求。上一节讲述的所有例子与配置在这里也都同样适用。但与浏览器不同的是，提交的文件和简单的表单字段，客户端发送的数据可以更加复杂，数据可以指定为某种特定的内容类型（content type）——比如，一个多路上传请求可能第一部分是个文件，而第二部分是个JSON格式的数据：

```
POST /someUrl
Content-Type: multipart/mixed

--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
Content-Disposition: form-data; name="meta-data"
Content-Type: application/json; charset=UTF-8
Content-Transfer-Encoding: 8bit

{
    "name": "value"
}
--edt7Tfrdusa7r3lNQc79vXuhIIMlatb7PQg7Vp
Content-Disposition: form-data; name="file-data"; filename="file.properties"
Content-Type: text/xml
Content-Transfer-Encoding: 8bit
... File Data ...
```

对于名称为 `meta-data` 的部分，你可以通过控制器方法上的 `@RequestParam("meta-data") String metadata` 参数来获得。但对于那部分请求体中为JSON格式数据的请求，你可能更想通过接受一个对应的强类型对象，就像 `@RequestBody` 通过 `HttpMessageConverter` 将一般请求的请求体转换成一个对象一样。

这是可能的，你可以使用 `@RequestPart` 注解来实现，而非 `@RequestParam` 。该注解将使得特定多路请求的请求体被传给 `HttpMessageConverter` ，并且在转换时考虑多路请求中不同的内容类型参数 `'Content-Type'` ：

```
@RequestMapping(path = "/someUrl", method = RequestMethod.POST)
public String onSubmit(@RequestPart("meta-data") MetaData metadata, @RequestPart("file
-data") MultipartFile file) {

    // ...

}
```

请注意 `MultipartFile` 方法参数是如何能够在 `@RequestParam` 或 `@RequestPart` 注解下互用的，两种方法都能拿到数据。但，这里的方法参数 `@RequestPart("meta-data") MetaData` 则会因为请求中的内容类型请求头 `'Content-Type'` 被读入成为JSON数据，然后再通过 `MappingJackson2HttpMessageConverter` 被转换成特定的对象。

21.11 异常处理

- 处理器异常解析器HandlerExceptionHandler
- @ExceptionHandler注解
- 处理一般的Spring MVC异常
- 使用@ResponseStatus注解业务异常
- Servlet默认容器错误页面的定制化

21.11.1 处理器异常解析器HandlerExceptionHandler

Spring的处理器异常解析器`HandlerExceptionResolver`接口的实现负责处理各类控制器执行过程中出现的异常。某种程度上讲，`HandlerExceptionResolver`与你在web应用描述符`web.xml`文件中能定义的异常映射（exception mapping）很相像，不过它比后者提供了更灵活的方式。比如它能提供异常被抛出时正在执行的是哪个处理器这样的信息。并且，一个更灵活（programmatic）的异常处理方式可以为你提供更多选择，使你在请求被直接转向到另一个URL之前（与你使用Servlet规范的异常映射是一样的）有更多的方式来处理异常。

实现`HandlerExceptionResolver`接口并非实现异常处理的唯一方式，它只是提供了`resolveException(Exception, Hanlder)`方法的一个实现而已，方法会返回一个`ModelAndView`。除此之外，你还可以框架提供的`SimpleMappingExceptionResolver`或在异常处理方法上注解`@ExceptionHandler`。`SimpleMappingExceptionResolver`允许你获取可能抛出的异常类的名字，并把它映射到一个视图名上去。这与Servlet API提供的异常映射特性是功能等价的，但你也可以基于此实现粒度更精细的异常映射。而`@ExceptionHandler`注解的方法则会在异常抛出时被调用以处理该异常。这样的方法可以定义在`@Controller`注解的控制器类里，也可以定义在`@ControllerAdvice`类中，后者可以使该异常处理方法被应用到更多的`@Controller`控制器中。下一小节将提供更为详细的信息。

21.11.2 @ExceptionHandler注解

`HandlerExceptionResolver` 接口以及 `SimpleMappingExceptionResolver` 解析器类的实现使得你能声明式地将异常映射到特定的视图上，还可以在异常被转发（forward）到对应的视图前使用Java代码做些判断和逻辑。不过在一些场景，特别是依靠 `@ResponseBody` 返回响应而非依赖视图解析机制的场景下，直接设置响应的状态码并将客户端需要的错误信息直接写回响应体中，可能是更方便的方法。

你也可以使用 `@ExceptionHandler` 方法来做到这点。如果 `@ExceptionHandler` 方法是在控制器内部定义的，那么它会接收并处理由控制器（或其任何子类）中的 `@RequestMapping` 方法抛出的异常。如果你将 `@ExceptionHandler` 方法定义在 `@ControllerAdvice` 类中，那么它会处理相关控制器中抛出的异常。下面的代码就展示了一个定义在控制器内部的 `@ExceptionHandler` 方法：

```
@Controller
public class SimpleController {

    // @RequestMapping methods omitted ...

    @ExceptionHandler(IOException.class)
    public ResponseEntity<String> handleIOException(IOException ex) {
        // prepare responseEntity
        return responseEntity;
    }

}
```

此外， `@ExceptionHandler` 注解还可以接受一个异常类型的数组作为参数值。若抛出了已在列表中声明的异常，那么相应的 `@ExceptionHandler` 方法将会被调用。如果没有给注解任何参数值，那么默认处理的异常类型将是方法参数所声明的那些异常。

与标准的控制器 `@RequestMapping` 注解处理方法一样， `@ExceptionHandler` 方法的方法参数和返回值也可以很灵活。比如，在Servlet环境下方法可以接收 `HttpServletRequest` 参数，而Portlet环境下方法可以接收 `PortletRequest` 参数。返回值可以是 `String` 类型——这种情况下会被解析为视图名——可以是 `ModelAndView` 类型的对象，也可以是 `ResponseEntity` 。或者你还可以在方法上添加 `@ResponseBody` 注解以使用消息转换器会转换信息为特定类型的数据，然后把它们写回到响应流中。

21.11.3 处理一般的Spring MVC异常

处理请求的过程中，Spring MVC可能会抛出一些的异常。 `SimpleMappingExceptionResolver` 可以根据需要很方便地将任何异常映射到一个默认的错误视图。但，如果客户端是通过自动检测响应的方式来分发处理异常的，那么后端就需要为响应设置对应的状态码。根据抛出异常的类型不同，可能需要设置不同的状态码来标识是客户端错误（4xx）还是服务器端错误（5xx）。

默认处理器异常解析器 `DefaultHandlerExceptionResolver` 会将Spring MVC抛出的异常转换成对应的错误状态码。该解析器在MVC命名空间配置或MVC Java配置的方式下默认已经被注册了，另外，通过 `DispatcherServlet` 注册也是可行的（即不使用MVC命名空间或Java编程方式进行配置的时候）。下表列出了该解析器能处理的一些异常，及他们对应的状态码。

异常	HTTP状态码
`BindException`	400 (无效请求)
`ConversionNotSupportedException`	500 (服务器内部错误)
`HttpMediaTypeNotAcceptableException`	406 (不接受)
`HttpMediaTypeNotSupportedException`	415 (不支持的媒体类型)
`HttpMessageNotReadableException`	400 (无效请求)
`HttpMessageNotWritableException`	500 (服务器内部错误)
`HttpRequestMethodNotSupportedException`	405 (不支持的方法)
`MethodArgumentNotValidException`	400 (无效请求)
`MissingServletRequestParameterException`	400 (无效请求)
`MissingServletRequestPartException`	400 (无效请求)
`NoHandlerFoundException`	404 (请求未找到)
`NoSuchRequestHandlingMethodException`	404 (请求未找到)
`TypeMismatchException`	400 (无效请求)
`MissingPathVariableException`	500 (服务器内部错误)
`NoHandlerFoundException`	404 (请求未找到)

以下待翻译。

The `DefaultHandlerExceptionResolver` works transparently by setting the status of the response. However, it stops short of writing any error content to the body of the response while your application may need to add developer- friendly content to every error response

for example when providing a REST API. You can prepare a `ModelAndView` and render error content through view resolution -- i.e. by configuring a `ContentNegotiatingViewResolver`, `MappingJackson2JsonView`, and so on. However, you may prefer to use `@ExceptionHandler` methods instead.

If you prefer to write error content via `@ExceptionHandler` methods you can extend `ResponseEntityExceptionHandler` instead. This is a convenient base for `@ControllerAdvice` classes providing an `@ExceptionHandler` method to handle standard Spring MVC exceptions and return `ResponseEntity`. That allows you to customize the response and write error content with message converters. See the `ResponseEntityExceptionHandler` javadocs for more details.

21.11.4 使用@ResponseStatus注解业务异常

业务异常可以使用`@ResponseStatus`来注解。当异常被抛出时，`ResponseStatusExceptionResolver`会设置相应的响应状态码。`DispatcherServlet`会默认注册一个`ResponseStatusExceptionResolver`以供使用。

21.11.5 Servlet默认容器错误页面的定制化

当响应的状态码被设置为错误状态码，并且响应体中没有内容时，Servlet容器通常会渲染一个HTML错误页。若需要定制容器默认提供的错误页，你可以在 `web.xml` 中定义一个错误页面 `<error-page>` 元素。在Servlet 3规范出来之前，该错误页元素必须被显式指定映射到一个具体的错误码或一个异常类型。从Servlet 3开始，错误页不再需要映射到其他信息了，这意味着，你指定的位置就是对Servlet容器默认错误页的自定制了。

```
<error-page>
    <location>/error</location>
</error-page>
```

这里错误页的位置所在可以是一个JSP页面，或者其他的一些URL，只要它指定容器里任意一个 `@Controller` 控制器下的处理器方法：

写回 `HttpServletResponse` 的错误信息和错误状态码可以在控制器中通过请求属性来获取：

```
@Controller
public class ErrorController {

    @RequestMapping(path = "/error", produces = MediaType.APPLICATION_JSON_UTF8_VALUE)
    @ResponseBody
    public Map<String, Object> handle(HttpServletRequest request) {

        Map<String, Object> map = new HashMap<String, Object>();
        map.put("status", request.getAttribute("javax.servlet.error.status_code"));
        map.put("reason", request.getAttribute("javax.servlet.error.message"));

        return map;
    }

}
```

或者在JSP中这么使用:

```
<%@ page contentType="application/json" pageEncoding="UTF-8"%>
{
    status:<%=request.getAttribute("javax.servlet.error.status_code") %>,
    reason:<%=request.getAttribute("javax.servlet.error.message") %>
}
```

21.12 Web安全

Spring Security项目为保护web应用免受恶意攻击提供了一些特性。你可以去看看参考文档的这几小节："CSRF保护"、"安全响应头"以及"Spring MVC集成"。不过并非应用的所有特性都需要引入Spring Security。比如，需要CSRF保护的话，你仅需要简单地在配置中添加一个过滤器 `CsrfFilter` 和处理器 `CsrfRequestDataValueProcessor` 。你可以参考Spring MVC Showcase一节，观看一个完整的展示。

另一个选择是使用其他专注于Web安全的框架。HDIV就是这样的一个框架，并且它能与Spring MVC良好集成。

21.13 "约定优于配置"的支持

对于许多项目来说，不打破已有的约定，对于配置等有可预测的默认值是非常适合的。现在，Spring MVC对 约定优于配置 这个实践已经有了显式的支持。这意味着什么呢？意味着如果你为项目选择了一组命名上的约定/规范，你将能减少 大量 的配置项，比如一些必要的处理器映射、视图解析器、 `ModelAndView` 实例的配置等。这能帮你快速地建立起项目原型，此外在某种程度上（通常是好的方面）维护了整个代码库的一致性should you choose to move forward with it into production.

具体来说，支持“约定优于配置”涉及到MVC的三个核心方面：模型、视图，和控制器。

21.13.1 控制器类名-处理器映射ControllerClassNameHandlerMapping

`ControllerClassNameHandlerMapping` 类是 `HandlerMapping` 接口的一个实现，它是通过一个约定来解析请求URL及处理该请求的 `@Controller` 控制器实例之间的映射关系。

请看下面一个简单的控制器实现。请注意留意该类的 名称：

```
public class **ViewShoppingCartController** implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {
        // 这个例子中方法的具体实现并不重要，故忽略。
    }

}
```

对应的Spring Web MVC配置文件如下所示：

```
<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>

<bean id="**viewShoppingCart**" class="x.y.z.ViewShoppingCartController">
    <!-- 注入需要的依赖 -->
</bean>
```

`ControllerClassNameHandlerMapping` 会查找当前应用上下文中注册的所有处理器（也即控制器）bean，并去除类名的 `Controller` 后缀作为决定处理器映射的依据。因此，类名 `ViewShoppingCartController` 会被映射到匹配 `/viewshoppingcart*` 的请求URL上。

让我们多看几个例子，这样你对于核心的思想会马上熟悉起来（注意URL中路径是全小写，而 `Controller` 控制器类名符合驼峰命名法）：

- `WelcomeController` 将映射到 `/welcome*` 请求URL
- `HomeController` 将映射到 `/home*` 请求URL
- `IndexController` 将映射到 `/index*` 请求URL
- `RegisterController` 将映射到 `/register*` 请求URL

对于 `MultiActionController` 处理器类，映射规则要稍微复杂一些。请看下面的代码，假设这里的控制器都是 `MultiActionController` 的实现：

- `AdminController` 将映射到 `/admin/*` 请求URL
- `CatalogController` 将映射到 `/catalog/*` 请求URL

只要所有控制器 `Controller` 实现都遵循 `xxxController` 这样的命名规范，那么 `ControllerClassNameHandlerMapping` 能把你从定义维护一个 长长长 `SimpleUrlHandlerMapping` 映射表的重复工作中拯救出来。

`ControllerClassNameHandlerMapping` 类继承自 `AbstractHandlerMapping` 基类。因此，你可以视它与其他 `HandlerMapping` 实现一样，定义你所需要的拦截器 `HandlerInterceptor` 实例及其他所有东西。

21.13.2 模型ModelMap(ModelAndView)

`ModelMap` 类其实就是一个豪华版的 `Map` ，它使得你为视图展示需要所添加的对象都遵循一个显而易见的约定被命名。请看下面这个 `Controller` 实现，并请注意，添加到 `ModelAndView` 中去的对象都没有显式地指定键名。

```
public class DisplayShoppingCartController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse
response) {

        List cartItems = // 拿到一个CartItem对象的列表
        User user = // 拿到当前购物的用户User

        ModelAndView mav = new ModelAndView("displayShoppingCart"); <-- 逻辑视图名

        mav.addObject(cartItems); <-- 啊哈，直接添加的对象，没有指定名称
        mav.addObject(user); <-- 啊哈再来一次

        return mav;
    }
}
```

`ModelAndView` 内部使用了一个 `ModelMap` 类，它是 `Map` 的一个实现，会自动为添加进来的对象生成一个键名。为添加对象生成名称的策略是，若添加对象是一个纯Java bean（a scalar object），比如 `User` ，那么使用对象类的短类名（short class name）来作为该对象的名称。下面是一些例子，展示了为添加到 `ModelMap` 实例中的纯Java对象所生成的名称：

- 添加一个 `x.y.User` 实例，为其生成的名称为 `user`
- 添加一个 `x.y.Registration` 实例，为其生成的名称为 `registration`
- 添加一个 `x.y.Foo` 实例，为其生成的名称为 `foo`
- 添加一个 `java.util.HashMap` 实例，为其生成的名称为 `hashMap` 。这种情况下，显式地声明一个键名可能更好，因为 `hashMap` 的约定并非那么符合直觉
- 添加一个 `null` 值将导致程序抛出一个 `IllegalArgumentException` 参数非法异常。若你所添加的（多个）对象有可能为 `null` 值，那你也需要显式地指定它（们）的名字

啥？键名不能自动变复数形式么？

Spring Web MVC的约定优于配置支持尚不能支持自动复数转换。这意思是，你不能期望往 `ModelAndView` 中添加一个 `Person` 对象的 `List` 列表时，框架会自动为其生成一个名称 `people` 。

这个决定是经过许多争论后的结果，最终“最小惊喜原则”胜出并为大家所接受。

为集合 `Set` 或列表 `List` 生成键名所采取的策略，是先检查集合的元素类型、拿到第一个对象的短类名，然后在其后面添加 `List` 作为名称。添加数组对象也是同理，尽管对于数组我们就不需再检查数组内容了。下面给出的几个例子可以阐释一些东西，让集合的名称生成语义变得更加清晰：

- 添加一个带零个或多个 `x.y.User` 元素类型的数组 `x.y.User[]` ，为其生成的键名是 `userList`
- 添加一个带零个或多个 `x.y.User` 元素类型的数组 `x.y.Foo[]` ，为其生成的键名是 `fooList`
- 添加一个带零个或多个 `x.y.User` 元素类型的数组 `java.util.ArrayList` ，为其生成的键名是 `userList`
- 添加一个带零个或多个 `x.y.Foo` 元素类型的数组 `java.util.HashSet` ，为其生成的键名是 `fooList`
- 一个 空的 `java.util.ArrayList` 则根本不会被添加

21.13.3 视图-请求与视图名的映射

`RequestToViewNameTranslator` 接口可以在逻辑视图名未被显式提供的情况下，决定一个可用的逻辑视图 `View` 名。

`DefaultRequestToViewNameTranslator` 能够将请求URL映射到逻辑视图名上去，如下面代码例子所示：

```
public class RegistrationController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {
        // 处理请求……
        ModelAndView mav = new ModelAndView();
        // 向Model中添加需要的数据
        return mav;
        // 请注意这里，没有设置任何View对象或逻辑视图名
    }

}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- 这个众人皆知的bean将为我们自动生成视图名 -->
    <bean id="viewNameTranslator" class="org.springframework.web.servlet.view.DefaultR
equestToViewNameTranslator"/>

    <bean class="x.y.RegistrationController">
        <!-- 如果需要，注入依赖 -->
    </bean>

    <!-- 请请求URL映射到控制器名 -->
    <bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandle
rMapping"/>

    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResour
ceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>

</beans>
```

请注意在 `handleRequest(...)` 方法实现中，返回的 `ModelAndView` 对象上自始至终未设置任何 `View` 对象或逻辑视图名。这是由 `DefaultRequestToViewNameTranslator` 完成的，它的任务就是从请求的URL中生成一个逻辑视图名。在上面的例子中， `RegistrationController` 与配置的 `ControllerClassNameHandlerMapping` 一起使用的结果是，一个URL为 `<http://localhost/registration.html>` 的请求，会经由 `DefaultRequestToViewNameTranslator` 生成并对应到一个逻辑视图名 `registration` 上。该逻辑视图名又会由 `InternalResourceViewResolver` bean解析到 `/WEB-INF/jsp/registration.jsp` 视图上。

> 你无需显式地定义一个 `DefaultRequestToViewNameTranslator` bean。如果默认的 `DefaultRequestToViewNameTranslator` 配置已能满足你的需求，那么你无需配置，Spring Web MVC的 `DispatcherServlet` 会为你实例化这样一个默认的对象。

当然，如果你需要更改默认的设置，那你就需要手动地配置自己的 `DefaultRequestToViewNameTranslator` bean。关于可配置属性的一些详细信息，你可以去咨询 `DefaultRequestToViewNameTranslator` 类详细的java文档。

21.14 HTTP缓存支持

一个好的HTTP缓存策略可以极大地提高一个web应用的性能及客户端的体验。谈到HTTP缓存，它主要是与HTTP的响应头 `'Cache-Control'` 相关，其次另外的一些响应头比如 `'Last-Modified'` 和 `'ETag'` 等也会起一定的作用。

HTTP的响应头 `'Cache-Control'` 主要帮助私有缓存（比如浏览器端缓存）和公共缓存（比如代理端缓存）了解它们应该如果缓存HTTP响应，以便后用。

ETag（实体标签）是一个HTTP响应头，可由支持HTTP/1.1的web应用服务器设置返回，主要用于标识给定的URL下的内容有无变化。可以认为它是 `Last-Modified` 头的一个更精细的后续版本。当服务器端返回了一个ETag头的资源表示时，客户端就可以在后续的GET请求中使用这个表示，一般是将它放在 `If-None-Match` 请求头中。此时若内容没有变化，服务器端会直接返回 `304: 内容未更改` 。

这一节将讲解其他一些在Spring Web MVC应用中配置HTTP缓存的方法。

21.14.1 HTTP请求头Cache-Control

Spring MVC提供了许多方式来配置"Cache-Control"请求头，支持在许多场景下使用它。关于该请求头完整详尽的所有用法，你可以参考RFC 7234的第5.2.2小节，这里我们只讲解最常用的几种用法。

Spring MVC的许多API中都使用了这样的惯例配置：`setCachePeriod(int seconds)`，若返回值为：

- `-1`，则框架不会生成一个`'Cache-Control'`缓存控制指令响应头
- `0`，则指示禁止使用缓存，服务器端返回缓存控制指令`'Cache-Control: no-store'`
- 任何`n > 0`的值，则响应会被缓存`n`秒，并返回缓存控制指令`'Cache-Control: max-age=n'`

`CacheControl`构造器类被简单的用来描述"Cache-Control"缓存控制指令，使你能更容易地创建自己的HTTP缓存策略。创建完了以后，`CacheControl`类的实例就可以在Spring MVC的许多API中被传入为方法参数了。

```
// 缓存一小时 - "Cache-Control: max-age=3600"
CacheControl ccCacheOneHour = CacheControl.maxAge(1, TimeUnit.HOURS);

// 禁止缓存 - "Cache-Control: no-store"
CacheControl ccNoStore = CacheControl.noStore();

// 缓存十天，对所有公共缓存和私有缓存生效
// 响应不能被公共缓存改变
// "Cache-Control: max-age=864000, public, no-transform"
CacheControl ccCustom = CacheControl.maxAge(10, TimeUnit.DAYS)
                                    .noTransform().cachePublic();
```

21.14.2 对静态资源的HTTP缓存支持

为优化站点性能，静态资源应该带有恰当的 `'Cache-Control'` 值与其他必要的头。配置一个 `ResourceHttpRequestHandler` 处理器服务静态资源请求不仅会读取文件的元数据并填充 `'Last-Modified'` 头的值，正确配置时 `'Cache-Control'` 头也会被填充。【这段翻得还不是很清晰】

你可以设置 `ResourceHttpRequestHandler` 上的 `cachePeriod` 属性值，或使用一个 `CacheControl` 实例来支持更细致的指令：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**")
                .addResourceLocations("/public-resources/")
                .setCacheControl(CacheControl.maxAge(1, TimeUnit.HOURS).cachePublic())
;
    }

}
```

XML中写法则如下：

```
<mvc:resources mapping="/resources/**" location="/public-resources/">
    <mvc:cache-control max-age="3600" cache-public="true"/>
</mvc:resources>
```

21.14.3 在控制器中设置Cache-Control、ETag和Last-Modified响应头

控制器能处理带有 `'Cache-Control'` 、 `'ETag'` 及/或 `'If-Modified-Since'` 头的请求，如果服务端在响应中设置了 `'Cache-Control'` 响应头，那么我们推荐在控制器内对这些请求头进行处理。这涉及一些工作：计算最后更改时间 `long` 和/或请求的ETag值、与请求头的 `'If-Modified-Since'` 值做比较，并且在资源未更改的情况下在响应中返回一个304（资源未更改）状态码。

正如在"使用HttpEntity"一节中所讲，控制器可以通过 `HttpEntity` 类与请求/响应交互。返回 `ResponseEntity` 的控制器可以在响应中包含HTTP缓存的信息，如下代码所示：

```
@RequestMapping("/book/{id}")
public ResponseEntity<Book> showBook(@PathVariable Long id) {

    Book book = findBook(id);
    String version = book.getVersion();

    return ResponseEntity
                .ok()
                .cacheControl(CacheControl.maxAge(30, TimeUnit.DAYS))
                .eTag(version) // 这里也能操作最后修改时间lastModified，只不过没有一一展示
                .body(book);
}
```

这样做不仅会在响应头中设置 `'ETag'` 及 `'Cache-Control'` 相关的信息，同时也会 尝试将响应状态码设置为 **`HTTP 304 Not Modified`** （资源未修改）及将响应体置空——如果客户端携带的请求头信息与控制器设置的缓存信息能够匹配的话。

如果希望在 `@RequestMapping` 方法上也能完成同样的事，那么你可以这样做：

```
@RequestMapping
public String myHandleMethod(WebRequest webRequest, Model model) {

    long lastModified = // 1. 应用相关的方式计算得到(application-specific calculation)

    if (request.checkNotModified(lastModified)) {
        // 2. 快速退出 — 不需要更多处理了
        return null;
    }

    // 3. 若资源更改了，那么再进行请求处理阶段，一般而言是准备响应内容
    model.addAttribute(...);
    return "myViewName";
}
```

这里最重要的两个地方是：调用 `request.checkNotModified(lastModified)` 方法，以及返回 `null` 。前者（方法调用）在返回 `true` 之前会将响应状态码设为304；而后者，在检查是否更改的方法调用返回 `true` 的基础上直接将方法返回，这会通知Spring MVC不再对请求做任何处理。

另外要注意的是，检查资源是否发生了更改有3种方式：

- `request.checkNotModified(lastModified)` 方法会将传入的参数值（最后修改时间）与请求头 `'If-Modified-Since'` 的值进行比较
- `request.checkNotModified(eTag)` 方法会将传入的参数值与请求头 `'ETag'` 的值进行比较
- `request.checkNotModified(eTag, lastModified)` 方法会同时进行以上两种比较。也即是说，只有在两个比较都被判定为未修改时，服务器才会返回一个304响应状态码 `HTTP 304 Not Modified` （资源未修改）

21.14.4 弱ETag（Shallow ETag)

对ETag的支持是由Servlet的过滤器 `ShallowEtagHeaderFilter` 提供的。它是纯Servlet技术实现的过滤器，因此，它可以与任何web框架无缝集成。 `ShallowEtagHeaderFilter` 过滤器会创建一个我们称为弱ETag（与强ETag相对，后面会详述）的对象。过滤器会将渲染的JSP页面的内容（包括其他类型的内容）缓存起来，然后以此生成一个MD5哈希值，并把这个值作为ETag头的值写回响应中。下一次客户端再次请求这个同样的资源时，它会将这个ETag的值写到 `If-None-Match` 头中。过滤器会检测到这个请求头，然后再次把视图渲染出来并比较两个哈希值。如果比较的结果是相同的，那么服务器会返回一个 `304` 。正如你所见，视图仍然会被渲染，因此本质上这个过滤器并非节省任何计算资源。唯一节省的东西，是带宽，因为被渲染的响应不会被整个发送回客户端。

请注意，这个策略节省的是网络带宽，而非CPU。因为对于每个请求，完整的响应仍然需要被整个计算出来。而其他在控制器层级实现的策略（上几节所述的）可以同时节省网络带宽及避免多余计算。

你可以在 `web.xml` 中配置 `ShallowEtagHeaderFilter` ：

```
<filter>
    <filter-name>etagFilter</filter-name>
    <filter-class>org.springframework.web.filter.ShallowEtagHeaderFilter</filter-class>

</filter>

<filter-mapping>
    <filter-name>etagFilter</filter-name>
    <servlet-name>petclinic</servlet-name>
</filter-mapping>
```

如果是在Servlet 3.0以上的环境下，可以这么做：

```
public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

    // ...

    @Override
    protected Filter[] getServletFilters() {
        return new Filter[] { new ShallowEtagHeaderFilter() };
    }

}
```

更多配置细节，请参考第21.15 基于代码的Servlet容器初始化一小节。

21.15 基于代码的Servlet容器初始化

在Servlet 3.0以上的环境下，你可以通过编程的方式来配置Servlet容器了。你可以完全放弃 `web.xml` ，也可以两种配置方式同时使用。以下是一个注册 `DispatcherServlet` 的例子：

```
import org.springframework.web.WebApplicationInitializer;

public class MyWebApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) {
        XmlWebApplicationContext appContext = new XmlWebApplicationContext();
        appContext.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

        ServletRegistration.Dynamic registration = container.addServlet("dispatcher",
new DispatcherServlet(appContext));
        registration.setLoadOnStartup(1);
        registration.addMapping("/");
    }

}
```

Spring MVC提供了一个 `WebApplicationInitializer` 接口，实现这个接口能保证你的配置能自动被检测到并应用于Servlet 3容器的初始化中。 `WebApplicationInitializer` 有一个实现，是一个抽象的基类，名字叫 `AbstractDispatcherServletInitializer` 。有了它，要配置 `DispatcherServlet` 将变得更简单，你只需要覆写相应的方法，在其中提供servlet映射、 `DispatcherServlet` 所需配置的位置即可：

```
public class MyWebAppInitializer extends AbstractAnnotationConfigDispatcherServletInit
ializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] { MyWebConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }

}
```

以上的例子适用于使用基于Java配置的Spring应用。如果你使用的是基于XML的Spring配置方式，那么请直接继承 `AbstractDispatcherServletInitializer` 这个类：

```
public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        return null;
    }

    @Override
    protected WebApplicationContext createServletApplicationContext() {
        XmlWebApplicationContext cxt = new XmlWebApplicationContext();
        cxt.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");
        return cxt;
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }

}
```

`AbstractDispatcherServletInitializer` 同样也提供了便捷的方式来添加过滤器 `Filter` 实例并使他们自动被映射到 `DispatcherServlet` 下：

```
public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

    // ...

    @Override
    protected Filter[] getServletFilters() {
        return new Filter[] { new HiddenHttpMethodFilter(), new CharacterEncodingFilte
r() };
    }

}
```

每个过滤器被添加时，默认的名称都基于其类类型决定，并且它们会被自动地映射到 `DispatcherServlet` 下。

关于异步支持， `AbstractDispatcherServletInitializer` 的保护方法 `isAsyncSupported` 提供了一个集中的地方来开关 `DispatcherServlet` 上的这个配置，它会对所有映射到这个分发器上的过滤器生效。默认情况下，这个标志被设为 `true` 。

最后，如果你需要对 `DispatcherServlet` 做进一步的定制，你可以覆写 `createDispatcherServlet` 这个方法。

21.16 配置Spring MVC

21.2.1 WebApplicationContext中特殊的bean类型小节和21.2.2 默认的DispatcherServlet配置小节解释了何谓Spring MVC的特殊bean，以及 `DispatcherServlet` 所使用的默认实现。在这小节中，你将了解配置Spring MVC的其他两种方式：MVC Java编程配置，以及MVC XML命名空间。

MVC Java编程配置和MVC命名空间都提供了相似的默认配置，以覆写 `DispatcherServlet` 的默认值。目标在于为大多数应用软件免去创建相同配置的麻烦，同时也想为配置Spring MVC提供一个更易理解的指南、一个简单的开始点，它只需要很少或不需任何关于底层配置的知识。

你可以选用MVC Java编程配置或MVC命名空间的方式，这完全取决于你的喜好。若你能读完后面的小节，你会发现使用MVC Java编程配置的方式能更容易看到底层具体的配置项，并且能对创建的Spring MVC bean有更细粒度的定制空间。不过，我们还是从头来看起吧。

21.16.1 启用MVC Java编程配置或MVC命名空间

要启用MVC Java编程配置，你需要在其中一个注解了 `@Configuration` 的类上添加 `@EnableWebMvc` 注解：

```
@Configuration
@EnableWebMvc
public class WebConfig {

}
```

要启用XML命名空间，请在你的DispatcherServlet上下文中（如果没有定义任何DispatcherServlet上下文，那么就在根上下文中）添加一个 `mvc:annotation-driven` 元素：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven/>

</beans>
```

上面的简单的声明代码，就已经默认注册了一个 `RequestMappingHandlerMapping` 、一个 `RequestMappingHandlerAdapter` ，以及一个 `ExceptionHandlerExceptionResolver` ，以支持对使用了 `@RequestMapping` 、 `@ExceptionHandler` 及其他注解的控制器方法的请求处理。

同时，上面的代码还启用了以下的特性：

1. Spring 3风格的类型转换支持。这是使用一个配置的转换服务ConversionService实例，以及the JavaBeans PropertyEditors used for Data Binding.
2. 使用 `@NumberFormat` 对数字字段进行格式化，类型转换由 `ConversionService` 实现
3. 使用 `@DateTimeFormat` 注解对 `Date` 、 `Calendar` 、 `Long` 及Joda Time类型的字段进行格式化
4. 使用 `@Valid` 注解对 `@Controller` 输入进行验证——前提是classpath路径下比如提供符合JSR-303规范的验证器

5. HTTP消息转换 `HttpMessageConverter` 的支持，对注解了 `@RequestMapping` 或 `@ExceptionHandler` 方法的 `@RequestBody` 方法参数或 `@ResponseBody` 返回值生效

下面给出了一份由 `mvc:annotation-driven` 注册可用的HTTP消息转换器的完整列表：

1. 转换字节数组的 `ByteArrayHttpMessageConverter`
2. 转换字符串的 `StringHttpMessageConverter`
3. `ResourceHttpMessageConverter` ： `org.springframework.core.io.Resource` 与所有媒体类型之间的互相转换
4. `SourceHttpMessageConverter` ：从（到） `javax.xml.transform.Source` 的转换
5. `FormHttpMessageConverter` ：数据与 `MultiValueMap<String, String>` 之间的互相转换
6. `Jaxb2RootElementHttpMessageConverter` ：Java对象与XML之间的互相转换——该转换器在classpath路径下有JAXB2依赖并且没有Jackson 2 XML扩展时被注册
7. `MappingJackson2HttpMessageConverter` ：从（到）JSON的转换——该转换器在classpath下有Jackson 2依赖时被注册
8. `MappingJackson2XmlHttpMessageConverter` ：从（到）XML的转换——该转换器在classpath下有Jackson 2 XML扩展时被注册
9. `AtomFeedHttpMessageConverter` ：Atom源的转换——该转换器在classpath路径下有Rome时被注册
10. `RssChannelHttpMessageConverter` ：RSS源的转换——该转换器在classpath路径下有Rome时被注册

你可以参考21.16.12 消息转换器一小节，了解如何进一步定制这些默认的转换器。

> Jackson JSON和XML转换器是通过 `Jackson2ObjectMapperBuilder` 创建的 `ObjectMapper` 实例创建的，目的在于提供更好的默认配置
>
> 该builder会使用以下的默认属性对Jackson进行配置：
>
> 1. 禁用 `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES`
> 2. 禁用 `MapperFeature.DEFAULT_VIEW_INCLUSION`
>
> 同时，如果检测到在classpath路径下存在这些模块，该builder也会自动地注册它们：
>
> 1. jackson-datatype-jdk7: 支持Java 7的一些类型，例如 `java.nio.file.Path`
> 2. jackson-datatype-joda: 支持Joda-Time类型
> 3. jackson-datatype-jsr310: 支持Java 8的Date & Time API类型
> 4. jackson-datatype-jdk8: 支持Java 8其他的一些类型，比如 `Optional` 等

21.16.2 默认配置的定制化

在MVC Java编程配置方式下，如果你想对默认配置进行定制，你可以自己实现`WebMvcConfigurer`接口，要么继承`WebMvcConfigurerAdapter`类并覆写你需要定制的方法：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    // Override configuration methods...

}
```

在MVC XML命名空间下，如果你想对默认配置进行定制，请查看`<mvc:annotation-driven/>`元素支持的属性和子元素。你可以查看Spring MVC XML schema，或使用IDE的自动补全功能来查看有哪些属性和子元素是可以配置的。

21.16.3 转换与格式化

数字的 `Number` 类型和日期 `Date` 类型的格式化是默认安装了的，包括 `@NumberFormat` 注解和 `@DateTimeFormat` 注解。如果classpath路径下存在Joda Time依赖，那么完美支持Joda Time的时间格式化库也会被安装好。如果要注册定制的格式化器或转换器，请覆写 `addFormatters` 方法：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addFormatters(FormatterRegistry registry) {
        // Add formatters and/or converters
    }

}
```

使用MVC命名空间时， `<mvc:annotation-driven>` 也会进行同样的默认配置。要注册定制的格式化器和转换器，只需要提供一个转换服务 `ConversionService` ：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven conversion-service="conversionService"/>

    <bean id="conversionService"
            class="org.springframework.format.support.FormattingConversionServiceFacto
ryBean">
        <property name="converters">
            <set>
                <bean class="org.example.MyConverter"/>
            </set>
        </property>
        <property name="formatters">
            <set>
                <bean class="org.example.MyFormatter"/>
                <bean class="org.example.MyAnnotationFormatterFactory"/>
            </set>
        </property>
        <property name="formatterRegistrars">
            <set>
                <bean class="org.example.MyFormatterRegistrar"/>
            </set>
        </property>
    </bean>

</beans>
```

> 关于如何使用格式化管理器FormatterRegistrar，请参考 8.6.4 FormatterRegistrar SPI一节，以及 `FormattingConversionServiceFactoryBean` 的文档。

21.16.4 验证

Spring提供了一个验证器Validator接口，应用的任何一层都可以使用它来做验证。在Spring MVC中，你可以配置一个全局的 `Validator` 实例，用以处理所有注解了 `@Valid` 的元素或注解了 `@Validated` 的控制器方法参数、以及/或在控制器内的 `@InitBinder` 方法中用作局部的 `Validator` 。全局验证器与局部验证器实例可以结合起来使用，提供组合验证。

Spring还支持JSR-303/JSR-349的Bean验证。这是通过 `LocalValidatorFactoryBean` 类实现的，它为Spring的验证器接口 `org.springframework.validation.Validator` 到Bean验证的 `javax.validation.Validator` 接口做了适配。这个类可以插入到Spring MVC的上下文中，作为一个全局的验证器，如下所述。

如果在classpath下存在Bean验证器，诸如Hibernate Validator等，那么 `@EnableWebMvc` 或 `<mvc:annotation-driven>` 默认会自动使用 `LocalValidatorFactoryBean` 为Spring MVC应用提供Bean验证的支持。

> 有时，能将 `LocalValidatorFactoryBean` 直接注入到控制器或另外一个类中会更方便。
>
> Sometimes it's convenient to have a `LocalValidatorFactoryBean` injected into a controller or another class. The easiest way to do that is to declare your own `@Bean` and also mark it with `@Primary` in order to avoid a conflict with the one provided with the MVC Java config.
>
> If you prefer to use the one from the MVC Java config, you'll need to override the `mvcValidator` method from `WebMvcConfigurationSupport` and declare the method to explicitly return `LocalValidatorFactory` rather than `Validator`. See Section 21.16.13, "Advanced Customizations with MVC Java Config" for information on how to switch to extend the provided configuration.

此外，你也可以配置你自己的全局 `Validator` 验证器实例：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public Validator getValidator(); {
        // return "global" validator
    }

}
```

XML中做法如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd">

    <mvc:annotation-driven validator="globalValidator"/>

</beans>
```

若要同时使用全局验证和局部验证，只需添加一个（或多个）局部验证器即可：

```
@Controller
public class MyController {

    @InitBinder
    protected void initBinder(WebDataBinder binder) {
        binder.addValidators(new FooValidator());
    }

}
```

做完这个最少的配置之后，任何时候只要方法中有参数注解了 `@Valid` 或 `@Validated` ，配置的验证器就会自动对它们做验证。任何无法通过的验证都会被自动报告为错误并添加到 `BindingResult` 对象中去，你可以在方法参数中声明它并获取这些错误，同时这些错误也能在Spring MVC的HTML视图中被渲染。

21.16.5 拦截器

你可以配置处理器拦截器 `HandlerInterceptors` 或web请求拦截器 `WebRequestInterceptors` 等拦截器，并配置它们拦截所有进入容器的请求，或限定到符合特定模式的URL路径。

在MVC Java编程配置下注册拦截器的方法：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new LocaleInterceptor());
        registry.addInterceptor(new ThemeInterceptor()).addPathPatterns("/**").exclude
PathPatterns("/admin/**");
        registry.addInterceptor(new SecurityInterceptor()).addPathPatterns("/secure/*"
);
    }

}
```

在MVC XML命名空间下，则使用 `<mvc:interceptors>` 元素：

```
<mvc:interceptors>
    <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor"/>
    <mvc:interceptor>
        <mvc:mapping path="/**"/>
        <mvc:exclude-mapping path="/admin/**"/>
        <bean class="org.springframework.web.servlet.theme.ThemeChangeInterceptor"/>
    </mvc:interceptor>
    <mvc:interceptor>
        <mvc:mapping path="/secure/*"/>
        <bean class="org.example.SecurityInterceptor"/>
    </mvc:interceptor>
</mvc:interceptors>
```

21.16.6 内容协商

You can configure how Spring MVC determines the requested media types from the request. The available options are to check the URL path for a file extension, check the "Accept" header, a specific query parameter, or to fall back on a default content type when nothing is requested. By default the path extension in the request URI is checked first and the "Accept" header is checked second.

The MVC Java config and the MVC namespace register `json` , `xml` , `rss` , `atom` by default if corresponding dependencies are on the classpath. Additional path extension-to-media type mappings may also be registered explicitly and that also has the effect of whitelisting them as safe extensions for the purpose of RFD attack detection (see the section called "Suffix Pattern Matching and RFD" for more detail).

Below is an example of customizing content negotiation options through the MVC Java config:

```
_@Configuration_
_@EnableWebMvc_
public class WebConfig extends WebMvcConfigurerAdapter {

    _@Override_
    public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {
        configurer.mediaType("json", MediaType.APPLICATION_JSON);
    }
}
```

In the MVC namespace, the `<mvc:annotation-driven>` element has a `content- negotiation-manager` attribute, which expects a `ContentNegotiationManager` that in turn can be created with a `ContentNegotiationManagerFactoryBean` :

```
<mvc:annotation-driven content-negotiation-manager="contentNegotiationManager"/>

<bean id="contentNegotiationManager" class="org.springframework.web.accept.ContentNego
tiationManagerFactoryBean">
    <property name="mediaTypes">
        <value>
            json=application/json
            xml=application/xml
        </value>
    </property>
</bean>
```

If not using the MVC Java config or the MVC namespace, you'll need to create an instance of `ContentNegotiationManager` and use it to configure `RequestMappingHandlerMapping` for request mapping purposes, and `RequestMappingHandlerAdapter` and `ExceptionHandlerExceptionResolver` for content negotiation purposes.

Note that `ContentNegotiatingViewResolver` now can also be configured with a `ContentNegotiationManager`, so you can use one shared instance throughout Spring MVC.

In more advanced cases, it may be useful to configure multiple `ContentNegotiationManager` instances that in turn may contain custom `ContentNegotiationStrategy` implementations. For example you could configure `ExceptionHandlerExceptionResolver` with a `ContentNegotiationManager` that always resolves the requested media type to `"application/json"`. Or you may want to plug a custom strategy that has some logic to select a default content type (e.g. either XML or JSON) if no content types were requested.

21.16.7 视图控制器

以下的一段代码相当于定义一个 `ParameterizableViewController` 视图控制器的快捷方式，该控制器会立即将一个请求转发（forwards）给一个视图。请确保仅在以下情景下才使用这个类：当控制器除了将视图渲染到响应中外不需要执行任何逻辑时。

以下是一个例子，展示了如何在MVC Java编程配置方式下将所有 `"/"` 请求直接转发给名字为 `"home"` 的视图：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("home");
    }

}
```

在MVC XML命名空间下完成同样的配置，则使用 `<mvc:view-controller>` 元素：

```
<mvc:view-controller path="/" view-name="home"/>
```

21.16.8 视图解析器

MVC提供的配置简化了视图解析器的注册工作。

以下的代码展示了在MVC Java编程配置下，如何为内容协商配置FreeMarker HTML模板和Jackson作为JSON数据的默认视图解析：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.enableContentNegotiation(new MappingJackson2JsonView());
        registry.jsp();
    }

}
```

在MVC XML命名空间下实现相同配置：

```
<mvc:view-resolvers>
    <mvc:content-negotiation>
        <mvc:default-views>
            <bean class="org.springframework.web.servlet.view.json.MappingJackson2Json
View"/>
        </mvc:default-views>
    </mvc:content-negotiation>
    <mvc:jsp/>
</mvc:view-resolvers>
```

需要注意的是，使用FreeMarker, Velocity, Tiles, Groovy Markup及script模板作为视图技术时，仍需要配置一些其他选项。

MVC命名空间为每种视图都提供了相应的元素。比如下面代码是FreeMarker需要的配置：

```
<mvc:view-resolvers>
    <mvc:content-negotiation>
        <mvc:default-views>
            <bean class="org.springframework.web.servlet.view.json.MappingJackson2Json
View"/>
        </mvc:default-views>
    </mvc:content-negotiation>
    <mvc:freemarker cache="false"/>
</mvc:view-resolvers>

<mvc:freemarker-configurer>
    <mvc:template-loader-path location="/freemarker"/>
</mvc:freemarker-configurer>
```

在MVC Java编程配置方式下，添加一个视图对应的“配置器”bean即可：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) {
        registry.enableContentNegotiation(new MappingJackson2JsonView());
        registry.freeMarker().cache(false);
    }

    @Bean
    public FreeMarkerConfigurer freeMarkerConfigurer() {
        FreeMarkerConfigurer configurer = new FreeMarkerConfigurer();
        configurer.setTemplateLoaderPath("/WEB-INF/");
        return configurer;
    }

}
```

资源的服务

21.16.10 回到默认的Servlet来进行资源服务

这些配置允许你将 `DispatcherServlet` 映射到"/"路径（也即覆盖了容器默认Servlet的映射），但依然保留容器默认的Servlet以处理静态资源的请求。这可以通过配置一个URL映射到"/**"的处理器 `DefaultServletHttpRequestHandler` 来实现，并且该处理器在其他所有URL映射关系中优先级应该是最低的。

该处理器会将所有请求转发（forward）到默认的Servlet，因此需要保证它在所有URL处理器映射 `HandlerMappings` 的最后。如果你是通过 `<mvc:annotation-driven>` 的方式进行配置，或自己定制了 `HandlerMapping` 实例，那么你需要确保该处理器 `order` 属性的值比 `DefaultServletHttpRequestHandler` 的次序值 `Integer.MAXVALUE` 小。

使用默认的配置启用该特性，你可以：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer config
urer) {
        configurer.enable();
    }

}
```

XML命名空间只需一行：

```
    <mvc:default-servlet-handler/>
```

不过需要注意，覆写了"/"的Servlet映射后，默认Servlet的 `RequestDispatcher` 就必须通过名字而非路径来取得了。 `DefaultServletHttpRequestHandler` 会尝试在容器初始化的时候自动检测默认Servlet，这里它使用的是一份主流Servlet容器（包括Tomcat、Jetty、GlassFish、JBoss、Resin、WebLogic，和WWebSphere）已知的名称列表。如果默认Servlet被配置了一个其他的名字，或者使用了一个列表里未提供默认Servlet名称的容器，那么默认Servlet的名称必须被显式指定。正如下面代码所示：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
        configurer.enable("myCustomDefaultServlet");
    }

}
```

XML命名空间的配置方式：

```
<mvc:default-servlet-handler default-servlet-name="myCustomDefaultServlet"/>
```

21.16.11 路径匹配

这些配置允许你对许多与URL映射和路径匹配有关的设置进行定制。关于所有可用的配置选项，请参考PathMatchConfigurer类的API文档。

下面是采用MVC Java编程配置的一段代码：

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configurePathMatch(PathMatchConfigurer configurer) {
        configurer
            .setUseSuffixPatternMatch(true)
            .setUseTrailingSlashMatch(false)
            .setUseRegisteredSuffixPatternMatch(true)
            .setPathMatcher(antPathMatcher())
            .setUrlPathHelper(urlPathHelper());
    }

    @Bean
    public UrlPathHelper urlPathHelper() {
        //...
    }

    @Bean
    public PathMatcher antPathMatcher() {
        //...
    }

}
```

在XML命名空间下实现同样的功能，可以使用 `<mvc:path-matching>` 元素：

```
<mvc:annotation-driven>
    <mvc:path-matching
        suffix-pattern="true"
        trailing-slash="false"
        registered-suffixes-only="true"
        path-helper="pathHelper"
        path-matcher="pathMatcher"/>
</mvc:annotation-driven>

<bean id="pathHelper" class="org.example.app.MyPathHelper"/>
<bean id="pathMatcher" class="org.example.app.MyPathMatcher"/>
```

21.16.12 消息转换器

使用MVC Java编程配置方式时，如果你想替换Spring MVC提供的默认转换器，完全定制自己的 `HttpMessageConverter` ，这可以通过覆写 `configureMessageConverters()` 方法来实现。如果你只是想定制一下，或者想在默认转换器之外再添加其他的转换器，那么可以通过覆写 `extendMessageConverters()` 方法来实现。

下面是一段例子，它使用定制的 `ObjectMapper` 构造了新的Jackson的JSON和XML转换器，并用它们替换了默认提供的转换器：

```
    @Configuration
    @EnableWebMvc
    public class WebConfiguration extends WebMvcConfigurerAdapter {

        @Override
        public void configureMessageConverters(List<HttpMessageConverter<?>> converter
s) {
            Jackson2ObjectMapperBuilder builder = new Jackson2ObjectMapperBuilder()
                    .indentOutput(true)
                    .dateFormat(new SimpleDateFormat("yyyy-MM-dd"))
                    .modulesToInstall(new ParameterNamesModule());
            converters.add(new MappingJackson2HttpMessageConverter(builder.build()));
            converters.add(new MappingJackson2XmlHttpMessageConverter(builder.xml().bu
ild()));
        }

    }
```

在上面的例子中， `Jackson2ObjectMapperBuilder` 用于为 `MappingJackson2HttpMessageConverter` 和 `MappingJackson2XmlHttpMessageConverter` 转换器创建公共的配置，比如启用tab缩进、定制的日期格式，并注册了一个模块jackson-module-parameter-names用于获取参数名（Java 8新增的特性）

> 除了 `jackson- dataformat-xml` ，要启用Jackson XML的tab缩进支持，还需要一个 `woodstox-core-asl` 依赖。

还有其他有用的Jackson模块可以使用：

1. jackson-datatype-money：提供了对 `javax.money` 类型的支持（非官方模块）
2. jackson-datatype-hibernate：提供了Hibernate相关的类型和属性支持（包含懒加载aspects）

在XML做同样的事也是可能的：

```
<mvc:annotation-driven>
    <mvc:message-converters>
        <bean class="org.springframework.http.converter.json.MappingJackson2HttpMessag
eConverter">
            <property name="objectMapper" ref="objectMapper"/>
        </bean>
        <bean class="org.springframework.http.converter.xml.MappingJackson2XmlHttpMess
ageConverter">
            <property name="objectMapper" ref="xmlMapper"/>
        </bean>
    </mvc:message-converters>
</mvc:annotation-driven>

<bean id="objectMapper" class="org.springframework.http.converter.json.Jackson2ObjectM
apperFactoryBean"
      p:indentOutput="true"
      p:simpleDateFormat="yyyy-MM-dd"
      p:modulesToInstall="com.fasterxml.jackson.module.paramnames.ParameterNamesModule"
/>

<bean id="xmlMapper" parent="objectMapper" p:createXmlMapper="true"/>
```

21.16.13 使用MVC Java编程进行高级定制

从上面许多例子你可以看到，MVC Java编程配置和MVC命名空间的方式都提供了更高抽象层级的应用配置，它不需要你对底下创建的bean有非常深入的了解，相反，这使得你能仅专注于应用需要的配置。不过，有时你可能希望对应用的更精细控制，或你就是单纯希望理解底下的配置和机制。

要做到更精细的控制，你要做的第一步就是看看底层都为你创建了哪些bean。若你使用MVC Java编程的方式进行配置，你可以看看java文档，以及 `WebMvcConfigurationSupport` 类的 `@Bean` 方法。这个类有的配置都会自动被 `@EnableWebMvc` 注解导入。事实上，如果你打开 `@EnableWebMvc` 的声明，你就会看到应用于其上的 `@Import` 注解。

精细控制的下一步是选择一个 `WebMvcConfigurationSupport` 创建的bean，定制它的属性，或你可以提供自己的一个实例。这确保做到以下两步：移除 `@EnableWebMvc` 注解以避免默认配置被自动导入，然后继承 `DelegatingWebMvcConfiguration` 类，它是 `WebMvcConfigurationSupport` 的一个子类。以下是一个例子：

```
@Configuration
public class WebConfig extends DelegatingWebMvcConfiguration {

    @Override
    public void addInterceptors(InterceptorRegistry registry){
        // ...
    }

    @Override
    @Bean
    public RequestMappingHandlerAdapter requestMappingHandlerAdapter() {
        // 自己创建适配器，或者调用super让基类处理
        // 然后在这里定制bean的一些属性
    }

}
```

> 应用应该只有一个继承 `DelegatingWebMvcConfiguration` 的配置类，或只有一个 `@EnableWebMvc` 注解的类，因为它们背后注册的bean都是相同的。
>
> 使用这个方式修改bean的属性，与这节前面展示的任何高抽象层级的配置方式并不冲突。 `WebMvcConfigurerAdapter` 的子类和 `WebMvcConfigurer` 的实现都还是会被使用。

21.16.14 使用MVC命名空间进行高级的定制化

如果使用MVC命名空间，要在默认配置的基础上实现粒度更细的控制，则要比使用MVC Java编程配置的方式难一些。

如果你确实需要这么做，那也尽量不要复制默认提供的配置，请尝试配置一个 `BeanPostProcessor` 后置处理器，用它来检测你要定制的bean。可以通过bean的类型来找，找到以后再修改需要定制的属性值。比如这样：

```
@Component
public class MyPostProcessor implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(Object bean, String name) throws BeanException {
        if (bean instanceof RequestMappingHandlerAdapter) {
            // 修改适配器的属性
        }
    }

}
```

注意，`MyPostProcessor` 需要被包含在 `<component scan/>` 的路径下，这样它才能被自动检测到；或者你也可以手动显式地用一个XML的bean定义来声明它。